Java Operators

Last Updated : 13 Dec, 2024

Java operators are special symbols that perform operations on variables or values. They can be classified into several categories based on their functionality. These operators play a crucial role in performing arithmetic, logical, relational, and bitwise operations etc.

Example: Here, we are using + and – operators.

```
// Java program to show the use of + and - operators
public class Geeks
{
    public static void main(String[] args)
    {

      // Declare and initialize variables
        int num1 = 500;
        int num2 = 100;

        // Using the + (addition) operator
        int sum = num1 + num2;
        System.out.println("The Sum is: "+sum);

        // Using the - (subtraction) operator
        int diff = num1 - num2;
        System.out.println("The Difference is: "+diff);

    }
}
```

Output

```
The Sum is: 600
The Difference is: 400
```

Types of Operators in Java

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

Let's see all these operators one by one with their proper examples.

1. Arithmetic Operators

Arithmetic Operators are used to perform simple arithmetic operations on primitive and non-primitive data types.

- ***** : Multiplication
- **/** : Division
- **%** : Modulo
- **+** : Addition
- **–** : Subtraction

Example:

```
// Java Program to show the use of
// Arithmetic Operators
import java.io.*;

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

        // Arithmetic operators on integers
        int a = 10;
        int b = 3;

        // Arithmetic operators on Strings
        String n1 = "15";
        String n2 = "25";

        // Convert Strings to integers
        int a1 = Integer.parseInt(n1);
        int b1 = Integer.parseInt(n2);
```

```
        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("a1 + b1 = " + (a1 + b1));

    }
}
```

Output

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
a1 + b1 = 40
```

2. Unary Operators

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

- `-` , Negates the value.
- `+` , Indicates a positive value (automatically converts `byte`, `char`, or `short` to `int`).
- `++` , Increments by 1.
    - **Post-Increment**: Uses value first, then increments.
    - **Pre-Increment**: Increments first, then uses value.
- `--` , Decrements by 1.
    - **Post-Decrement**: Uses value first, then decrements.
    - **Pre-Decrement**: Decrements first, then uses value.
- `!` , Inverts a boolean value.

Example:

```
// Java Program to show the use of
// Unary Operators
import java.io.*;

// Driver Class
```

```
class Geeks {
      // main function
    public static void main(String[] args)
    {
        // Interger declared
        int a = 10;
        int b = 10;

        // Using unary operators
        System.out.println("Postincrement : " + (a++));
        System.out.println("Preincrement : " + (++a));

        System.out.println("Postdecrement : " + (b--));
        System.out.println("Predecrement : " + (--b));
    }
}
```

Output

```
Postincrement : 10
Preincrement : 12
Postdecrement : 10
Predecrement : 8
```

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

The general format of the assignment operator is:

variable = value;

In many cases, the assignment operator can be combined with others to create shorthand compound statements. For example, `a += 5` replaces `a = a + 5`. Common compound operators include:

- `+=` , Add and assign.
- `-=` , Subtract and assign.
- `*=` , Multiply and assign.
- `/=` , Divide and assign.

- `%=` , Modulo and assign.

Example:

```java
// Java Program to show the use of
// Assignment Operators
import java.io.*;

// Driver Class
class Geeks {
    // Main Function
    public static void main(String[] args)
    {

        // Assignment operators
        int f = 7;
        System.out.println("f += 3: " + (f += 3));
        System.out.println("f -= 2: " + (f -= 2));
        System.out.println("f *= 4: " + (f *= 4));
        System.out.println("f /= 3: " + (f /= 3));
        System.out.println("f %= 2: " + (f %= 2));
        System.out.println("f &= 0b1010: " + (f &= 0b1010));
        System.out.println("f |= 0b1100: " + (f |= 0b1100));
        System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
        System.out.println("f <<= 2: " + (f <<= 2));
        System.out.println("f >>= 1: " + (f >>= 1));
        System.out.println("f >>>= 1: " + (f >>>= 1));
    }
}
```

Output

```
f += 3: 10
f -= 2: 8
f *= 4: 32
f /= 3: 10
f %= 2: 0
f &= 0b1010: 0
f |= 0b1100: 12
f ^= 0b1010: 6
f <<= 2: 24
f >>= 1: 12
f >>>= 1: 6
```

4. Relational Operators

Relational Operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is ,

variable ***relation_operator*** *value*

Relational operators compare values and return boolean results:

- `==` , Equal to.
- `!=` , Not equal to.
- `<` , Less than.
- `<=` , Less than or equal to.
- `>` , Greater than.
- `>=` , Greater than or equal to.

Example:

```java
// Java Program to show the use of
// Relational Operators
import java.io.*;

// Driver Class
class Geeks {
    // main function
    public static void main(String[] args)
    {
        // Comparison operators
        int a = 10;
        int b = 3;
        int c = 5;

        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 == c: " + (a == c));
        System.out.println("a != c: " + (a != c));
    }
}
```

Output

```
a > b: true
a < b: false
```

```
a >= b: true
a <= b: false
a == c: false
a != c: true
```

5. Logical Operators

Logical Operators are used to perform “logical AND” and “logical OR” operations, similar to AND gate and OR gate in digital electronics. They have a short-circuiting effect, meaning the second condition is not evaluated if the first is false.

Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

Example:

```java
// Java Program to show the use of
// Logical operators
import java.io.*;

class Geeks {

      // Main Function
    public static void main (String[] args) {

        // Logical operators
        boolean x = true;
        boolean y = false;

        System.out.println("x && y: " + (x && y));
        System.out.println("x || y: " + (x || y));
        System.out.println("!x: " + (!x));
    }
}
```

Output

```
x && y: false
x || y: true
!x: false
```

6. Ternary operator

The Ternary Operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary. The 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 ':'.

Example:

```
// Java program to illustrate
// max of three numbers using
// ternary operator.

public class Geeks {

    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
```

7. Bitwise Operators

Bitwise Operators are used to perform the manipulation of individual bits of a number and with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

- `&` **(Bitwise AND)** – returns bit-by-bit AND of input values.
- `|` **(Bitwise OR)** – returns bit-by-bit OR of input values.
- `^` **(Bitwise XOR)** – returns bit-by-bit XOR of input values.

- **~ (Bitwise Complement)** – inverts all bits (one's complement).

Example:

```
// Java Program to show the use of
// bitwise operators
import java.io.*;

class Geeks
{
    public static void main(String[] args)
    {
        // Bitwise operators
        int d = 0b1010;
        int e = 0b1100;

        System.out.println("d & e : " + (d & e));
        System.out.println("d | e : " + (d | e));
        System.out.println("d ^ e : " + (d ^ e));
        System.out.println("~d : " + (~d));
        System.out.println("d << 2 : " + (d << 2));
        System.out.println("e >> 1 : " + (e >> 1));
        System.out.println("e >>> 1 : " + (e >>> 1));
    }
}
```

Output

```
d & e : 8
d | e : 14
d ^ e : 6
~d : -11
d << 2 : 40
e >> 1 : 6
e >>> 1 : 6
```

8. 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. The general format ,

*number **shift_op** number_of_places_to_shift;*

- `<<` **(Left shift)** – Shifts bits left, filling 0s (multiplies by a power of two).
- `>>` **(Signed right shift)** – Shifts bits right, filling 0s (divides by a power of two), with the leftmost bit depending on the sign.
- `>>>` **(Unsigned right shift)** – Shifts bits right, filling 0s, with the leftmost bit always 0.

Example:

```
// Java Program to show the use of
// shift operators
import java.io.*;

class Geeks
{
    public static void main(String[] args)
    {
        int a = 10;

        // Using left shift
        System.out.println("a<<1 : " + (a << 1));

        // Using right shift
        System.out.println("a>>1 : " + (a >> 1));
    }
}
```

Output

```
a<<1 : 20
a>>1 : 5
```

9. instanceof operator

The instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. The general format ,

object ***instance of*** *class/subclass/interface*

```
// Java program to show the use of
// Instance of operator
public class Geeks
{
```

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

        Person obj1 = new Person();
        Person obj2 = new Boy();

        // As obj is of type person, it is not an
        // instance of Boy or interface
        System.out.println("obj1 instanceof Person: "
                           + (obj1 instanceof Person));
        System.out.println("obj1 instanceof Boy: "
                           + (obj1 instanceof Boy));
        System.out.println("obj1 instanceof MyInterface: "
                           + (obj1 instanceof MyInterface));

        // Since obj2 is of type boy,
        // whose parent class is person
        // and it implements the interface Myinterface
        // it is instance of all of these classes
        System.out.println("obj2 instanceof Person: "
                           + (obj2 instanceof Person));
        System.out.println("obj2 instanceof Boy: "
                           + (obj2 instanceof Boy));
        System.out.println("obj2 instanceof MyInterface: "
                           + (obj2 instanceof MyInterface));
    }
}

// Classes and Interfaces used
// are declared here
class Person {
}

class Boy extends Person implements MyInterface {
}

interface MyInterface {
}
```

Output

```
obj1 instanceof Person: true
obj1 instanceof Boy: false
obj1 instanceof MyInterface: false
obj2 instanceof Person: true
obj2 instanceof Boy: true
obj2 instanceof MyInterface: true
```

Precedence and Associativity of Java 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 showing the lowest precedence.

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ~ ! (type)	Right to left	Unary prefix
* / %	Left to right	Multiplicative
+ -	Left to right	Additive
<< >> >>>	Left to right	Shift
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Interesting Questions About Java Operators

1. Precedence and Associativity

There is often confusion when it comes to hybrid equations which are 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. The explanation of the below program is well written in comments within the program itself.

```
public class GFG
{
    public static void main(String[] args)
    {
        int a = 20, b = 10, c = 0;
          int 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 precedence 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
```

2. Be a Compiler

The compiler in our systems uses a lex tool to match the greatest match when generating tokens. This creates a bit of a problem if overlooked. For example, consider the statement **a=b+++c**; too many of the readers might seem to create a compiler error. But this statement is absolutely correct as the token created by lex is a, =, b, ++, +, c. Therefore, this statement has a similar effect of first assigning b+c to a and then incrementing b. Similarly, a=b+++++c; would generate an error as the tokens generated are a, =, b, ++, ++, +, c. which is actually an error as there is no operand after the second unary operand.

```
public class GFG
{
    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
```

3. Using + over ()

When using the + 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 a string first producing a string, and string objects concatenate when using +. Therefore it can create unwanted results.

```
public class GFG
{
    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
```

Advantages of Operators

The advantages of using operators in Java are mentioned below:

1. **Expressiveness**: Operators in Java provide a concise and readable way to perform complex calculations and logical operations.
2. **Time-Saving:** Operators in Java save time by reducing the amount of code required to perform certain tasks.