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. In this article, we will learn about Java Operators and learn all their types.

What are the Java Operators?

Operators in Java are the symbols used for performing specific operations in Java. Operators make tasks like addition, multiplication, etc which look easy although the implementation of these tasks is quite complex.

Types of Operators in Java

There are multiple types of operators in Java all are mentioned below:

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

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

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

Example:

- Java

```
// Java Program to implement
// Arithmetic Operators
import java.io.*;

// Drive Class
class GFG {
      // Main Function
    public static void main (String[] args) {
        // Arithmetic operators
        int a = 10;
        int b = 3;

        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));

    }
}
```

Output

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

2. Unary Operators

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

- – : **Unary minus**, used for negating the values.
- + : **Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- ++ : **Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
  - **Post-Increment:** Value is first used for computing the result and then incremented.
  - **Pre-Increment:** Value is incremented first, and then the result is computed.
- – – : **Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
  - **Post-decrement:** Value is first used for computing the result and then decremented.

    - **Pre-Decrement: The value** is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

Example:

- Java

```java
// Java Program to implement
// Uniary Operators
import java.io.*;

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

        // Using uniary 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 other operators to build a shorter version of the statement called a **Compound Statement**. For example, instead of a = a+5, we can write a **+=** 5.

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

Example:

- Java

```
// Java Program to implement
// Assignment Operators
import java.io.*;

// Driver Class
class GFG {
    // 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

These 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
```

Some of the relational operators are-

- **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
- **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
- **<, less than:** returns true if the left-hand side is less than the right-hand side.

- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

- Java

```java
// Java Program to implement
// Relational Operators
import java.io.*;

// Driver Class
class GFG {
    // 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

These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

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
// Java Program to implemenet

// Logical operators

import java.io.*;


// Driver Class

class GFG {

      // 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

```java
// 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
```

7. Bitwise Operators

These operators are used to perform the 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 the Binary indexed trees.

- **&, 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 complement representation of the input value, i.e., with all bits inverted.

- Java

```java
// Java Program to implement

// bitwise operators

import java.io.*;


// Driver class

class GFG {

    // main function

    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

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 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 the initial number. Similar effect to 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

```java
// Java Program to implement
// shift operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    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 the 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. General format-
`object instance of class/subclass/interface`

- Java

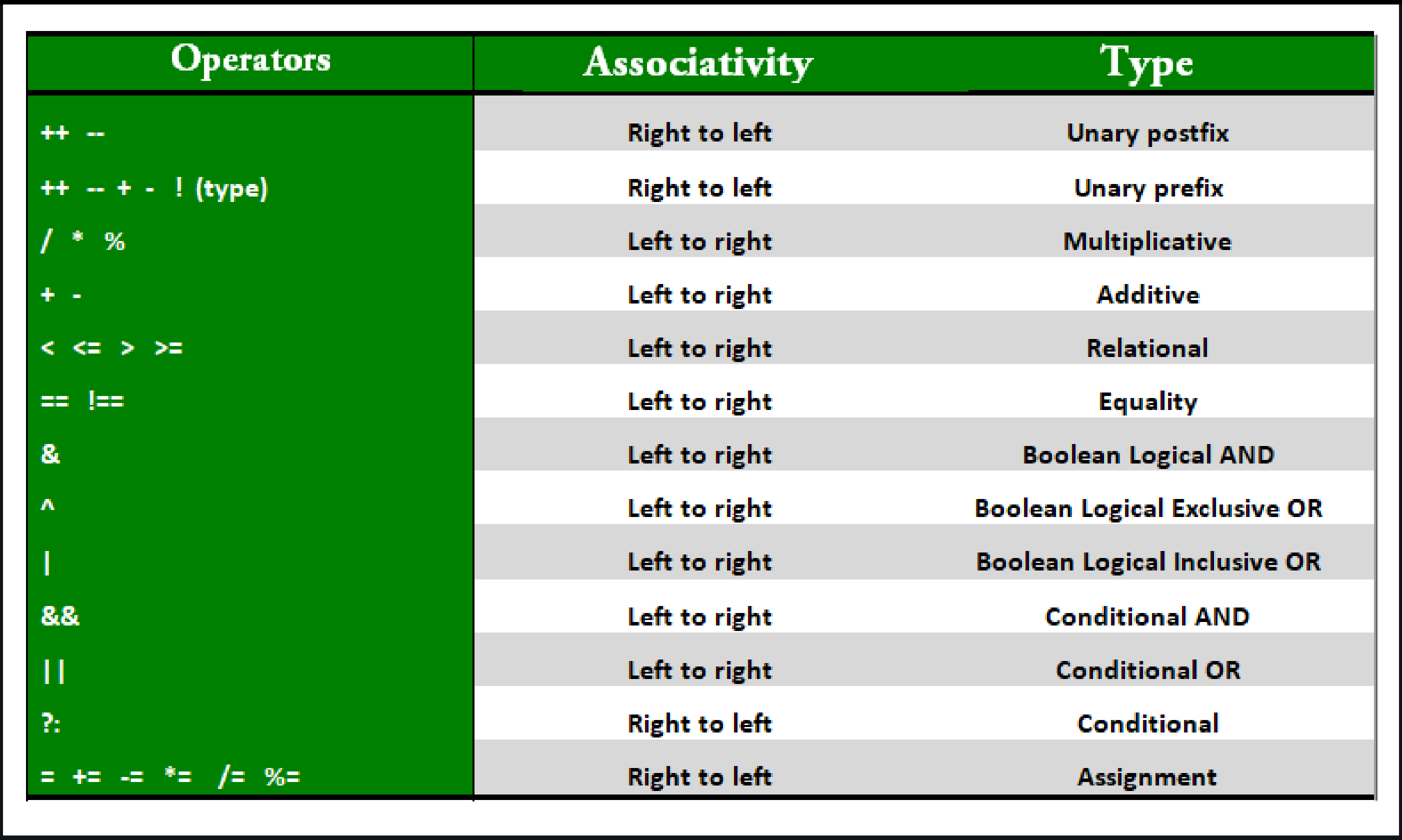
```java
// Java program to illustrate
// instance of operator

class operators {
    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));
    }
}

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	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.

- Java

```java
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 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.

- Java

```java
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
```

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.

- Java

```java
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
```

Advantages of Operators in Java

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.
3. **Improved Performance**: Using operators can improve performance because they are often implemented at the hardware level, making them faster than equivalent Java code.

Disadvantages of Operators in Java

The disadvantages of Operators in Java are mentioned below:

1. **Operator Precedence:** Operators in Java have a defined precedence, which can lead to unexpected results if not used properly.
2. **Type Coercion**: Java performs implicit type conversions when using operators, which can lead to unexpected results or errors if not used properly.
3. **Overloading**: Java allows for operator overloading, which can lead to confusion and errors if different classes define the same operator with different behavior.

FAQs in Java Operators

1. What is operators in Java with example?

Operators are the special symbols that are used for performing certain operations. For example, ‘+’ is used for addition where 5+4 will return the value 9.