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. Some of the types are:

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

Let's take a look at them in detail.

1. Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

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

```java
public class GfG {
    public static void main(String[] args) {
        int x = 10, y = 20;

        // Addition
        System.out.println("Addition: " + (x + y));

        // Subtraction
        System.out.println("Subtraction: " + (x - y));
        // Multiplication
        System.out.println("Multiplication: " + (x * y));

        // Division
        System.out.println("Division: " + (x / y));

        // Modulo
        System.out.println("Modulo: " + (x % y));
    }
}
```

```
Output
Addition: 30
Subtraction: -10
Multiplication: 200
Division: 0
Modulo: 10
```

Explanation:

- Each operator performs the specified arithmetic operation.
- Division (/) truncates the decimal portion when used with integers.

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

3. Assignment Operator: '=' Assignment operator is used to assigning a value to any variable. It has a 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 left operand with right operand and then assigning it to the variable on the left.
- -=, for subtracting right operand from left operand and then assigning it to the variable on the left.

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

```java
public class GfG {
    public static void main(String[] args) {
        int x = 10, y = 5;

        // Simple assignment
        x = y; // x is now 5
        System.out.println("After assignment: " + x);

        // Compound assignments
        x += y; // x = x + y
        System.out.println("After addition assignment: " + x);

        x -= y; // x = x - y
        System.out.println("After subtraction assignment: " + x);

        x *= y; // x = x * y
        System.out.println("After multiplication assignment: " + x);

        x /= y; // x = x / y
        System.out.println("After division assignment: " + x);
    }
}
```

Output

```
After assignment: 5
After addition assignment: 10
After subtraction assignment: 5
After multiplication assignment: 25
After division assignment: 5
```

Explanation:

- +=, -=, *=, and /= combine arithmetic and assignment in one operation.
- These are shorter forms of expressions like x = x + y.

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.

```java
public class GfG {
    public static void main(String[] args) {
        int x = 10, y = 20;

        // Equal to
        System.out.println("Equal to: " + (x == y));
        // Not equal to
        System.out.println("Not equal to: " + (x != y));

        // Greater than
        System.out.println("Greater than: " + (x > y));

        // Less than
        System.out.println("Less than: " + (x < y));

        // Greater than or equal to
        System.out.println("Greater than or equal to: " + (x >= y));

        // Less than or equal to
        System.out.println("Less than or equal to: " + (x <= y));
    }
}
```

```
Output
Equal to: false
Not equal to: true
Greater than: false
Less than: true
Greater than or equal to: false
Less than or equal to: true
```

Explanation:

- Relational operators return boolean values (true or false) based on the comparison.
- These operators are often used in loops and conditional statements.

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

```java
public class GfG {
    public static void main(String[] args) {
        boolean a = true, b = false;

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

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

        // Logical NOT
        System.out.println("Logical NOT: " + (!a));

        // Short-circuiting example
        int x = 10, y = 5;
        if (x > 5 && y < 10) {
            System.out.println("Both conditions are true.");
        }
    }
}
```

```
Output
Logical AND: false
Logical OR: true
Logical NOT: false
Both conditions are true.
```

Explanation:

- && ensures both conditions are true.
- || checks if at least one condition is true.
- ! negates the boolean value.
- Short-circuiting skips the second condition if the first condition determines the result.

6. Ternary operator: 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 ':'.

```
// Java program to illustrate
// max of three numbers using
// ternary operator.
public class GfG {
    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.

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 as 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.

Precedence and Associativity of Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first, as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude, with the top representing the highest precedence and the bottom 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 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
public class GfG {
    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: 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 are 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 tokens generated are a, =, b, ++, ++, +, c. which is actually an error as there is no operand after the second unary operand.

```java
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 + 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.

```
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
```