**Operators in Java**

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

1. **Arithmetic Operators:** They are used to perform simple arithmetic operations on primitive data types.
   * **\* :**Multiplication
   * **/ :**Division
   * **% :**Modulo
   * **+ :**Addition
   * **- :**Subtraction
2. **Unary Operators:** Unary operators need only one operand. They are used to increment, decrement, or negate a value.
   * **- :Unary minus** is used for negating the values.
   * **+ :Unary plus** is used for giving positive values. Only used when deliberately converting a negative value to a positive value.
   * **++ :Increment operator** is used for incrementing the value by 1. There are two varieties of increment operator.
     + **Post-Increment :**Value is first used for computing the result, and then it is incremented.
     + **Pre-Increment :**Value is incremented first, and then the result is computed.
   * **-- : Decrement operator** is used for decrementing the value by 1. There are two varieties of decrement operator.
     + **Post-decrement :**Value is first used for computing the result, and then it is decremented.
     + **Pre-Decrement :**Value is decremented first, and then the result is computed.
   * **! : Logical not operator** is used for inverting a boolean value.
3. **Assignment Operator : '='** Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e., the value given on the right hand side of the operator is assigned to the variable on the left; therefore, right hand side value must be declared before using it or it should be a constant. General format of assignment operator is,

variable **=** value;

1. In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a **=** a+5 , we can write a **+=** 5.
   * **+=**, for adding left operand with right operand and then assigning it to variable on the left.
   * **-=**, for subtracting left operand with right operand and then assigning it to variable on the left.
   * **\*=**, for multiplying left operand with right operand and then assigning it to variable on the left.
   * **/=**, for dividing left operand with right operand and then assigning it to variable on the left.
   * **%=**, for assigning modulo of left operand with right operand and then assigning it to variable on the left.
2. **Relational Operators :** These operators are used to check for relations such as equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as in conditional if else statements. General format is,

variable **relation\_operator** value

1. Some of the relational operators are:
   * **== , Equal to :**returns true of left hand side is equal to right hand side.
   * **!= , Not Equal to :**returns true of left hand side is not equal to right hand side.
   * **< , less than :**returns true of left hand side is less than right hand side.
   * **<= , less than or equal to :**returns true of left hand side is less than or equal to right hand side.
   * **> , Greater than :**returns true of left hand side is greater than right hand side.
   * **>= , Greater than or equal to:**returns true of left hand side is greater than or equal to right hand side.
2. **Logical Operators :** These operators are used to perform "logical AND" and "logical OR" operations, i.e., the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is that the second condition is not evaluated if the first one is false, i.e., it has short-circuiting effect. It is used extensively to test for several conditions for making a decision. Conditional operators are:
   * **&& , Logical AND :**returns true when both conditions are true.
   * **|| , Logical OR :**returns true if at least one condition is true.
3. **Ternary operator :** Ternary operator is a shorthand version of if-else statement. It has three operands, and hence the name ternary. General format is:

condition **?** if true **:** if false

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

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

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

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

Java

public **class** **operators** {

public static void main(String[] args)

{

// Initial values

int a = 2;

int b = 7;

System.out.println("a&b = " + (a & b)); // 0010 & 0111 = 0010 = 2

System.out.println("a|b = " + (a | b)); // 0010 | 0111 = 7

System.out.println("a^b = " + (a ^ b)); // 0010 ^ 0111 = 5

System.out.println("~a = " + ~a); // 2's complement of 2 = -3

}

}

**Output**

a&b = 2

a|b = 7

a^b = 5

~a = -3

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

number **shift\_op** number\_of\_places\_to\_shift;

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

**Precedence and Associativity of Operators**

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

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

**Interesting Questions on Operators**

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

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 precendence then associative

// rules are followed.

// e/f -> b\*d -> a+(b\*d) -> a+(b\*d)-(e/f)

System.out.println("a+b\*d-e/f = "+(a + b \* d - e / f));

}

}

**Output:**

a+b/d = 20

a+b\*d-e/f = 219

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

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

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

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