# Java Operators

Operators are symbols that perform operations on variables and values.

Operators in Java can be classified into 5 types:

Operators in Java can be classified into 5 types:

1.Arithmetic Operators

2.Assignment Operators

3.Relational Operators

4. Logical Operators

5.Unary Operators

6.Bitwise Operators

## 1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

|  |  |
| --- | --- |
| Operator | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division(Quotient after division) |
| % | Modulo Operation (Remainder after division) |

## 2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

int age;

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age.

Let's see some more assignment operators available in Java.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| = | a = b; | a = b; |
| += | a += b; | a = a + b; |
| -= | a -= b; | a = a - b; |
| \*= | a \*= b; | a = a \* b; |
| /= | a /= b; | a = a / b; |
| %= | a %= b; | a = a % b; |

### Example : Assignment Operators

class Main {

public static void main(String[] args) {

// create variables

int a = 4;

int var;

// assign value using =

var = a;

System.out.println("var using =: " + var);

// assign value using =+

var += a;

System.out.println("var using +=: " + var);

// assign value using =\*

var \*= a;

System.out.println("var using \*=: " + var);

}

}

**Output**

var using =: 4

var using +=: 8

var using \*=: 32

## 3. Java Relational Operators

Relational operators are used to check the relationship between two operands. For example,

// check is a is less than b

a < b;

Here, > operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Is Equal To | 3 == 5 returns false |
| != | Not Equal To | 3 != 5 returns true |
| > | Greater Than | 3 > 5 returns false |
| < | Less Than | 3 < 5 returns true |
| >= | Greater Than or Equal To | 3 >= 5 returns false |
| <= | Less Than or Equal To | 3 <= 5 returns false |

**Note**: Relational operators are used in decision making and loops

## 4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && (Logical AND) | expression1 **&&** expression2 | true only if both expression1 and expression2 are true |
| || (Logical OR) | expression1 **||** expression2 | true if either expression1 or expression2 is true |
| ! (Logical NOT) | **!**expression | true if expression is false and vice versa |

### Example : Logical Operators

class Main {

public static void main(String[] args) {

// && operator

System.out.println((5 > 3) && (8 > 5)); // true

System.out.println((5 > 3) && (8 < 5)); // false

// || operator

System.out.println((5 < 3) || (8 > 5)); // true

System.out.println((5 > 3) || (8 < 5)); // true

System.out.println((5 < 3) || (8 < 5)); // false

// ! operator

System.out.println(!(5 == 3)); // true

System.out.println(!(5 > 3)); // false

}

}

## 5. Java Unary Operators

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

|  |  |
| --- | --- |
| Operator | Meaning |
| + | **Unary plus**: not necessary to use since numbers are positive without using it |
| - | **Unary minus**: inverts the sign of an expression |
| ++ | **Increment operator**: increments value by 1 |
| -- | **Decrement operator**: decrements value by 1 |
| ! | **Logical complement operator**: inverts the value of a boolean |

## Increment and Decrement Operators

Java also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by **1**, while -- decrease it by **1**. For example,

int num = 5;

// increase num by 1

++num;

Here, the value of num gets increased to **6** from its initial value of **5**.

### Example : Increment and Decrement Operators

class Main {

public static void main(String[] args) {

// declare variables

int a = 12, b = 12;

int result1, result2;

// original value

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

// increment operator

result1 = ++a;

System.out.println("After increment: " + result1);

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

// decrement operator

result2 = --b;

System.out.println("After decrement: " + result2);

}

}

**Output**

Value of a: 12

After increment: 13

Value of b: 12

After decrement: 11

**++ and -- operator as prefix and postfix**

* If you use the ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value.
* If you use the ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1.

The -- operator works in a similar way like the ++ operator except it decreases the value by 1.

## 6. Java Bitwise Operators

Bitwise operators in Java are used to perform operations on individual bits. For example,

Bitwise complement Operation of 35

35 = 00100011 (In Binary)

~ 00100011

\_\_\_\_\_\_\_\_

11011100 = 220 (In decimal)

Here, ~ is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**).

The various bitwise operators present in Java are:

|  |  |
| --- | --- |
| Operator | Description |
| ~ | Bitwise Complement |
| << | Left Shift |
| >> | Right Shift |
| >>> | Unsigned Right Shift |
| & | Bitwise AND |
| ^ | Bitwise exclusive OR |

## Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00011101 = 29 (In decimal)

## Bitwise AND

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.

The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

\_\_\_\_\_\_\_\_

00001000 = 8 (In decimal)

## Bitwise Complement

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

\_\_\_\_\_\_\_\_

11011100 = 220 (In decimal)

### Example : Bitwise Complement

class Complement {

public static void main(String[] args) {

int number = 35, result;

result = ~number;

System.out.println(result);

}

}

When you run the program, the output will be:

-36

Why are we getting output -36 instead of 220?

It's because the compiler is showing 2's complement of that number; negative notation of the binary number.

For any integer n, 2's complement of n will be -(n+1).

Decimal Binary 2's complement

--------- --------- ---------------------------------------

0 00000000 -(11111111+1) = -00000000 = -0(decimal)

1 00000001 -(11111110+1) = -11111111 = -256(decimal)

12 00001100 -(11110011+1) = -11110100 = -244(decimal)

220 11011100 -(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

## Bitwise XOR

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

00001100

| 00011001

\_\_\_\_\_\_\_\_

00010101 = 21 (In decimal)

### Example : Bitwise XOR

class Xor {

public static void main(String[] args) {

int number1 = 12, number2 = 25, result;

result = number1 ^ number2;

System.out.println(result);

}

}

When you run the program, the output will be:

21

## Signed Left Shift

The left shift operator << shifts a bit pattern to the left by certain number of specified bits, and zero bits are shifted into the low-order positions.

212 (In binary: 11010100)

212 << 1 evaluates to 424 (In binary: 110101000)

212 << 0 evaluates to 212 (In binary: 11010100)

212 << 4 evaluates to 3392 (In binary: 110101000000)

## Signed Right Shift

The right shift operator >> shifts a bit pattern to the right by certain number of

specified bits.

212 (In binary: 11010100)

212 >> 1 evaluates to 106 (In binary: 01101010)

212 >> 0 evaluates to 212 (In binary: 11010100)

212 >> 8 evaluates to 0 (In binary: 00000000)

If the number is a 2's complement signed number, the sign bit is shifted into the high-order positions

.