Java Bitwise and Bit Shift Operators

Java provides 4 bitwise and 3 bit shift operators to perform bit operations. You will learn about them in detail in this article.

Table of Contents

Bitwise and bit shift operators are used on integral types (`byte` , `short` , `int` and `long`) to perform bit-level operations.

These operators are not commonly used. You will learn about a few use cases of bitwise operators in *Java enum type* chapter. This article will only focus on how these operators work.

There are 7 operators to perform bit-level operations in Java (4 bitwise and 3 bit shift).

Java Bitwise and Bit Shift Operators

Operator	Description
|	Bitwise OR
&	Bitwise AND
~	Bitwise Complement
^	Bitwise XOR
<<	Left Shift

>>	Right Shift
>>>	Unsigned Right Shift

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)
```

Example 1: Bitwise OR

```
class BitwiseOR {
    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:

```
29
```

Bitwise AND

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)
```

Example 2: Bitwise AND

```
class BitwiseAND {
    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:

```
8
```

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
```

Example 3: 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(deci
1                00000001       -(11111110+1) = -11111111 = -256(de
12               00001100       -(11110011+1) = -11110100 = -244(de
220             11011100       -(00100011+1) = -00100100 = -36(dec

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

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 4: 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)
```

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

        int number = 212, result;

        System.out.println(number << 1);
        System.out.println(number << 0);
        System.out.println(number << 4);
    }
}
```

When you run the program, the output will be:

```
424
212
3392
```

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.

Example 6: Signed Right Shift

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

        int number = 212, result;

        System.out.println(number >> 1);
        System.out.println(number >> 0);
        System.out.println(number >> 8);
```

When you run the program, the output will be:

```
106
212
0
```

Unsigned Right Shift

The unsigned right shift operator `<<` shifts zero into the leftmost position.

Example 7: Signed and UnSigned Right Shift

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

        int number1 = 5, number2 = -5;

        // Signed right shift
        System.out.println(number1 >> 1);

        // Unsigned right shift
        System.out.println(number1 >>> 1);

        // Signed right shift
        System.out.println(number2 >> 1);

        // Unsigned right shift
        System.out.println(number2 >>> 1);
     }
}
```

When you run the program, the output will be:

```
2
2
-3
2147483645
```

Notice, how signed and unsigned right shift works differently for 2's complement.

Java Tutorial