Bitwise Operators

Bitwise Operators in Programming

Introduction to Bitwise Operators

What Are Bitwise Operators?

- Bitwise operators perform operations directly on the binary representation of integers.
- Unlike arithmetic or logical operators, they manipulate individual bits of numbers.

Why Use Bitwise Operators?

- Efficient for low-level programming.
- Widely used in:
  - Cryptography.
  - Graphics programming.
  - Performance optimization.
  - Embedded systems.

Binary Number System Basics

- **Decimal**: Base-10 (0–9).
- **Binary**: Base-2 (0 and 1).
- Conversion examples:
  - $10_{10} = 1010_2$.
  - $7_{10} = 0111_2$.

Truth Table: Decimal to Binary

Decimal	Binary (4 Bits)	Explanation
0	0000	All bits are `0`.
1	0001	Least significant bit (LSB) is 1.
2	0010	Second bit from the right is 1.

3	0011 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0	Add the binary of `1` and `2` (`0001 + 0010`).
4	0100	Third bit from the right is `1`.
5	0101	Add `1` to `4` (`0100 + 0001`).
6	0110	Add `2` to `4` (`0100 + 0010`).
7	0111	Add `1`, `2`, and `4` (`0100 + 0010 + 0001`).
8	1000	Fourth bit from the right is `1`.
9	1001	Add `1` to `8` (`1000 + 0001`).
10	1010	Add `2` to `8` (`1000 + 0010`).
11	1011	Add `1`, `2`, and `8` (`1000 + 0010 + 0001`).
12	1100	Add `4` to `8` (`1000 + 0100`).
13	1101	Add `1`, `4`, and `8` (`1000 + 0100 + 0001`).
14	1110	Add `2`, `4`, and `8` (`1000 + 0100 + 0010`).
15	1111	All bits are `1`.

When discussing binary representations in Java, particularly for integers, **we always consider the 32-bit length** because Java uses a fixed-width representation for primitive data types:

- **`int`**: 32 bits (4 bytes).
- **`long`**: 64 bits (8 bytes).

Binary Representation in 32 Bits

- For **positive numbers**, the highest bit (most significant bit) is `0`.
- For **negative numbers**, the highest bit is `1` (due to **two's complement** representation).

For example:

1. **Positive 8**:
    - Decimal: `8`
    - Binary (32 bits): `00000000 00000000 00000000 00001000`.
2. **Negative -8**:
    - Decimal: `-8`
    - Binary (32 bits in two's complement): `11111111 11111111 11111111 11111000`.

    ### Conversion Steps

    1. Write the positive number in binary (`8 = 00000000 00000000 00000000 00001000`).
    2. Invert the bits. `11111111 11111111 11111111 11110111`
    3. Add `1` to the result.

       11111111 11111111 11111111 11110111

       + 00000000 00000000 00000000 00000001

       11111111 11111111 11111111 11111000

    -

Explanation of Binary Representation

1. **Binary System**:
    - Each bit represents a power of `2`, starting from the least significant bit (rightmost).
    - Example:
        - Binary `1010` = 1×2^3 + 0×2^2 + 1×2^1 + 0×2^0.
        - Result: 8+0+2+0=10(Decimal).

---

2. List of Bitwise Operators

Bitwise Operators

Operator	Symbol	Description
AND	&	Sets each bit to 1 if **both bits** are 1.
OR	|	Sets each bit to 1 if at **least one bit** is 1.
XOR	^	Sets each bit to 1 if **only one** bit is 1.
NOT	~	Inverts all the bits (1 becomes 0, 0 becomes 1).
Left Shift	<<	Shifts bits to the left, filling with 0s.
Right Shift	>>	Shifts bits to the right, sign-preserving.
Unsigned Right Shift	>>>	Shifts bits to the right, fills with 0.

Truth Tables for Operations

1. AND (&)

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

---

2. OR (|)

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

---

3. XOR (^)

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

---

4. NOT (~)

A	~A
0	1
1	0

Explanation of Operators

AND (&)

- **Rule**: The result is 1 if both bits are 1, otherwise `0`.
- **Example**:
    - 5 & 3 in binary:
        - 5=0101,
        - 3=0011,
        - Result: 0001(Decimal 1).

OR (|)

- **Rule**: The result is 1 if either bit is 1.
- **Example**:
    - 5 | 3:
        - 5=0101,
        - 3=0011
        - Result: 0111 (Decimal 7).
        - 1*2^1+1*2^0

XOR (^)

- **Rule**: The result is `1` if the bits differ, otherwise `0`.
- **Example**:
  - 5 ^ 35:
    - 5=0101,
    - 3=0011,
    - Result: 0110 (Decimal 6).

NOT (~)

- **Rule**: The result flips each bit.
- **Example**:
  - ~5:
    - 5= 0000 0000 0000 0000 0000 0000 0000 0101
    - Result: 1…….1111 1111 1010 (Decimal -6 in two's complement for signed integers).
      - Start with the number `5` (`0101` in binary).
      - Apply `~` to flip the bits: `1010`.
      - Interpret it as two's complement:
        - Flip the bits: `1010 → 0101`.
        - Add 1: `0101 + 0001 = 0110`.
        - Result: `-6`.
      - Thus, `~5 = -6` in Java.

Left Shift (<<)

The **left shift** operator **shifts the bits of a number to the left by a specified number of positions.** Zeros are added to the rightmost positions.

Formula:

$$\text{Value after left shift} = \text{number} \times 2^{\text{shift count}}$$

Example:

```
int num = 5;              // Binary: 0000 0101
int result = num << 2; // Shift left by 2 positions
// Binary: 0 ……0000 0001 0100 (Decimal: 20)

System.out.println(result); // Output: 20
```

Explanation:

- Binary of 5: `0000 0101`
- After shifting 2 positions to the left: `0001 0100` (which is `20` in decimal).

Right Shift (`>>`)

The **right shift** operator shifts the bits of a number to the right by a specified number of positions. The sign bit (the leftmost bit) is used to fill the empty positions:

- For **positive numbers**, zeros are filled.
- For **negative numbers**, ones are filled (sign extension).

Formula:

$$\text{Value after right shift} = \text{number} \div 2^{\text{shift count}}$$

Example (Positive Number):

int num = 20; // Binary: 0001 0100
int result = num >> 2; // Shift right by 2 positions
// Binary: 00 00 0101 (Decimal: 5)

System.out.println(result); // Output: 5

Example (Negative Number):

int num = -20; // Binary (Two's Complement): 1110 1100
int result = num >> 2; // Shift right by 2 positions
// Binary: 11 11 1011 (Decimal: -5)

System.out.println(result); // Output: -5

Explanation :

`1111 1011` has MSB = 1, so it's a negative number.

Find the Positive Equivalent

Invert All the Bits = 1111 1011 → 0000 0100

Add 1 to the Inverted Bits

```
  0000 0100
+ 0000 0001
___________
  0000 0101
```

the original number was negative (MSB = 1), add the negative sign: -5

- For `20`: Binary `0001 0100` becomes `0000 0101` (5 in decimal).
- For `-20`: Binary `1110 1100` becomes `1111 1011` (-5 in decimal).

Unsigned Right Shift (`>>>`)

The **unsigned right shift** operator shifts the bits of a number to the right by a specified number of positions. Unlike `>>`, it **always fills zeros** in the leftmost positions, regardless of whether the number is positive or negative.

Example (Positive Number):

int num = 20; // Binary: 0001 0100
int result = num >>> 2; // Shift right by 2 positions
// Binary: 0000 0101 (Decimal: 5)

System.out.println(result); // Output: 5

Example (Negative Number):

int num = -20; // Binary (Two's Complement): 1110 1100
int result = num >>> 2; // Shift right by 2 positions
// Binary: 0011 1011 (Decimal: 1073741819)

System.out.println(result); // Output: 1073741819

Explanation:

- For `20`: Same as `>>`, resulting in `0000 0101` (5 in decimal).
- For `-20`: The two's complement representation (`1110 1100`) shifts right and fills zeros: `0011 1011` (a large positive number).

Summary Table

Operator	Fills Left Bits	Sign Matters?	Example (Binary)	Result
`<<` (Left)	Zeros	No	`0000 0101 << 2`	`0001 0100` (20)

`>>` (Right)	Sign Bit	Yes	`1110 1100 >> 2`	`1111 1011` (-5)
`>>>` (Right)	Zeros	No	`1110 1100 >>> 2`	`0011 1011` (1073741819)

Key Points:

1. **`<<` (Left Shift)** multiplies by powers of 2.
2. **`>>` (Right Shift)** divides by powers of 2 while preserving the sign.
3. **`>>>` (Unsigned Right Shift)** divides by powers of 2 and always fills with zeros.