Understanding Bits and Bitwise Operators in Java

Before diving into **bitwise operators**, it's important to understand the concept of **bits** and how binary numbers (composed of bits) work. This will help you better understand how bitwise operations manipulate data.

What Are Bits?

A **bit** is the smallest unit of data in a computer and can only have two possible values: **0** or **1**. These values are also called **binary digits**.

In computer systems, all data, whether it's a number, text, or any other type of information, is represented as a sequence of bits.

For example:

- The number `5` in binary (base 2) is `101`.
- The number `3` in binary is `011`.

Each bit in a binary number represents a power of 2. Starting from the right, each bit represents increasing powers of 2.

For example:

- `5` in binary is `101` (which is `1 * 2^2 + 0 * 2^1 + 1 * 2^0`).
- `3` in binary is `011` (which is `0 * 2^2 + 1 * 2^1 + 1 * 2^0`).

Flipping Bits: `1` Becomes `0` and `0` Becomes `1`

One fundamental operation in bitwise manipulation is **inverting** bits, which means changing each `1` bit to `0` and each `0` bit to `1`. This is often referred to as a **bitwise NOT** operation and is performed using the `~` operator in Java.

Example:

If we take a binary number, say `5`, which is represented as `0000 0101` in an 8-bit system, and apply the bitwise NOT operator `~`, it will flip each bit:

```
Original number (5):  0000 0101
After bitwise NOT (~5): 1111 1010
```

In this example:

- The binary number `0000 0101` (which is 5 in decimal) becomes `1111 1010` (which is -6 in decimal, due to how negative numbers are represented in computers).

In binary, when bits are flipped, this can represent a **negative number** in a system using **two's complement**. This is a way for computers to represent negative numbers using bits.

Bitwise Operators in Java

Java provides a set of operators that directly manipulate the bits of integers. These **bitwise operators** perform operations on the binary representations of numbers.

1. AND (`&`)

The **AND** operator compares each corresponding pair of bits in two numbers. If both bits are `1`, the result is `1`; otherwise, it's `0`.

Example:

```
int x = 5;  // 0000 0101 in binary
int y = 3;  // 0000 0011 in binary
int result = x & y;  // 0000 0001 in binary, which is 1 in decimal
System.out.println(result);  // Output: 1
```

Here, we compare the bits of `5` (which is `0000 0101` in binary) and `3` (which is `0000 0011` in binary). The only position where both bits are `1` is the last one, so the result is `1`.

2. OR (`|`)

The **OR** operator compares each corresponding pair of bits in two numbers. If at least one of the bits is `1`, the result is `1`; otherwise, it's `0`.

Example:

```
int x = 5;  // 0000 0101 in binary
int y = 3;  // 0000 0011 in binary
int result = x | y;  // 0000 0111 in binary, which is 7 in decimal
System.out.println(result);  // Output: 7
```

Here, for every bit, if either `x` or `y` has a `1`, the result is `1`. Hence, `0000 0101 | 0000 0011 = 0000 0111`.

3. XOR (`^`)

The **XOR** (exclusive OR) operator compares each corresponding pair of bits. If exactly one of the bits is `1`, the result is `1`; otherwise, it's `0`.

Example:

```
int x = 5;  // 0000 0101 in binary
int y = 3;  // 0000 0011 in binary
int result = x ^ y;  // 0000 0110 in binary, which is 6 in decimal
System.out.println(result);  // Output: 6
```

Here, the XOR operation results in `1` only where the corresponding bits are different (i.e., one bit is `1` and the other is `0`).

4. Left Shift (`<<`)

The **left shift** operator shifts the bits of a number to the left by a specified number of positions. This operation multiplies the number by `2` for each shift.

Example:

```
int x = 5;  // 0000 0101 in binary
int result = x << 1;  // 0000 1010 in binary, which is 10 in decimal
System.out.println(result);  // Output: 10
```

Here, shifting `5` (which is `0000 0101`) one position to the left results in `10` (which is `0000 1010` in binary).

5. Right Shift (`>>`)

The **right shift** operator shifts the bits of a number to the right by a specified number of positions. This operation divides the number by `2` for each shift.

Example:

```
int x = 5;  // 0000 0101 in binary
int result = x >> 1;  // 0000 0010 in binary, which is 2 in decimal
System.out.println(result);  // Output: 2
```

Here, shifting `5` (which is `0000 0101`) one position to the right results in `2` (which is `0000 0010` in binary).

6. Bitwise NOT (`~`)

The **bitwise NOT** operator inverts each bit of a number. It changes `0` to `1` and `1` to `0`.

Example:

```
int x = 5;  // 0000 0101 in binary
int result = ~x;  // 1111 1010 in binary, which is -6 in decimal
System.out.println(result);  // Output: -6
```

In this example, the bits of `5` (which is `0000 0101`) are flipped to `1111 1010`, which is `-6` in decimal because of the way negative numbers are represented using two's complement.

Bitwise Operator Summary

- `&`: AND - Returns `1` only if both bits are `1`.
- `|`: OR - Returns `1` if at least one bit is `1`.
- `^`: XOR - Returns `1` if the bits are different.
- `<<`: Left shift - Shifts bits to the left (multiplies by powers of 2).
- `>>`: Right shift - Shifts bits to the right (divides by powers of 2).
- `~`: NOT - Inverts the bits.
