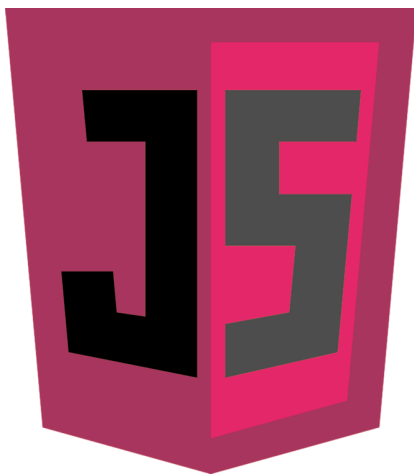


# Bitwise Operators

In this lesson, we will learn all about bitwise operators in JavaScript.

## WE'LL COVER THE FOLLOWING ^

- Some bitwise rules
  - Examples
  - Examples



## Bitwise Operators



Most programming languages define bitwise operators to allow expressions to play with bits. These operators generally use signed and unsigned integer numbers.


*JavaScript is a bit different.*

If any operands of the bitwise operators are not integer numbers, they are first converted into **32-bit** integers, and after the operation the result is converted back into a **64-bit** value, conforming with the **IEEE-754**

specification that is used to store Number values.

During these conversions, the `Number()` casting rules are applied. Infinite and NaN values are considered to be *zero* values.

JavaScript supports the following **bitwise operators**: bitwise NOT (`~`), bitwise AND (`&`), bitwise OR (`|`), bitwise XOR (`^`), left shift (`<<`), signed right shift (`>>`), and unsigned right shift (`>>>`).

 **NOTE:** The IEEE-754 representation uses the two's complement format to represent negative numbers.

## Some bitwise rules #

Bitwise NOT simply inverts all bits of its operand; 0 bits become 1, and vice versa. Bitwise AND, OR, and XOR combine the corresponding bits of their two operands to calculate the result.

For a single bit:

- bitwise **AND** results in 1 if and only if both bits are 1.
- bitwise **OR** produces 1 if any of the bits is 1.
- bitwise **XOR** produces 1 if and only if one of the bits is 0, while the other is 1.

## Examples #

Here are a few examples:

**JS** index.js

```
// 1111 1111 1111 1111 1111 1111 1110 1110 (-18)
// 0000 0000 0000 0000 0000 0000 0000 0010 (2)
// -----
// 0000 0000 0000 0000 0000 0000 0000 0010 (2)
console.log(-18 & 2); // 2

// 0000 0000 0000 0000 0000 0000 0000 1011 (11)
// 0000 0000 0000 0000 0000 0000 0001 0111 (23)
// -----
// 0000 0000 0000 0000 0000 0000 0001 1111 (31)
console.log(11 | 23.5); // 31
```



```
// 0000 0000 0000 0000 0000 0000 0000 1000 (8)
// 0000 0000 0000 0000 0000 0000 0000 0000 (0)
// -----
// 0000 0000 0000 0000 0000 0000 0000 0000 (0)
console.log(8 & null);

// 0000 0000 0000 0000 0000 0000 0001 0000 (16)
// 0000 0000 0000 0000 0000 0000 0000 1110 (14)
// -----
// 0000 0000 0000 0000 0000 0000 0001 1110 (30)
console.log(16 ^ 14);

// 0000 0000 0000 0000 0000 0000 0010 1010 (42)
// -----
// 1111 1111 1111 1111 1111 1111 1101 0101 (-43)
console.log(~"42");
```



The **left shift** operator shifts all bits in a number (first operand) to the left by the number of positions given in the second operand, i.e., to the right of the operator.

When the bits are shifted, the empty bits to the right of the number are filled with zeros to make the result a complete 32-bit number.

The **signed right shift** operator shifts all bits in a 32-bit number to the right while preserving the sign (positive or negative).

*A signed right shift is the exact opposite of a left shift.*

The **unsigned right shift** operator shifts all bits in a 32-bit number to the right. For numbers that are positive, the effect is the same as a signed right shift.

When the bits are shifted, the empty bits to the left of the number are filled with zeros to make the result a complete 32-bit number.

The shift operations don't modify the operand, if it is a variable.

# Examples #

Let's see a few examples:

js index.js

```
// 0000 0000 0000 0000 0000 0000 0000 1100 (12)
// -----
// 0000 0000 0000 0000 0000 0000 0011 0000 (48)
console.log(12 << 2); // 48

// 0000 0000 0000 0000 0000 0000 0001 1101 (29)
// -----
// 0000 0000 0000 0000 0000 0000 0000 0111 (7)
console.log(29 >> 2); // 7

// 0000 0000 0000 0000 0000 0000 0001 1101 (29)
// -----
// 0000 0000 0000 0000 0000 0000 0000 0111 (7)
console.log(29 >>> 2); // 7

// 1111 1111 1111 1111 1111 1111 1100 1100 (-52)
// -----
// 1111 1111 1111 1111 1111 1111 1111 1001 (-7)
console.log(-52 >> 3); // -7

// 1111 1111 1111 1111 1111 1111 1100 1100 (-52)
// -----
// 0001 1111 1111 1111 1111 1111 1111 1001 (536870905)
console.log(-52 >>> 3); // 536870905
```



In the *next lesson*, we'll meet the Assignment operators!