A table summarizing the basic JavaScript operators along with examples for each:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Operator** | **Description** | **Example** | **Result** |
| **Arithmetic** | **+** | Addition | **5 + 3** | **8** |
|  | **-** | Subtraction | **5 - 3** | **2** |
|  | **\*** | Multiplication | **5 \* 3** | **15** |
|  | **/** | Division | **10 / 2** | **5** |
|  | **%** | Modulus | **10 % 3** | **1** |
|  | **\*\*** | Exponentiation | **2 \*\* 3** | **8** |
| **Assignment** | **=** | Assignment | **let x = 10** | **x** is **10** |
|  | **+=** | Addition assignment | **x += 5** | **x** is **15** |
|  | **-=** | Subtraction assignment | **x -= 3** | **x** is **7** |
|  | **\*=** | Multiplication assignment | **x \*= 2** | **x** is **20** |
|  | **/=** | Division assignment | **x /= 2** | **x** is **5** |
|  | **%=** | Modulus assignment | **x %= 3** | **x** is **1** |
| **Comparison** | **==** | Equal | **5 == '5'** | **true** |
|  | **!=** | Not equal | **5 != '5'** | **false** |
|  | **===** | Strict equal | **5 === 5** | **true** |
|  | **!==** | Strict not equal | **5 !== '5'** | **true** |
|  | **>** | Greater than | **10 > 5** | **true** |
|  | **>=** | Greater than or equal | **10 >= 10** | **true** |
|  | **<** | Less than | **5 < 10** | **true** |
|  | **<=** | Less than or equal | **5 <= 5** | **true** |
| **Logical** | **&&** | Logical AND | **true && true** | **true** |
|  | ` |  | ` | Logical OR |
|  | **!** | Logical NOT | **!true** | **false** |
| **Bitwise** | **&** | AND | **5 & 1** | **1** |
|  | ` | ` | OR | `5 |
|  | **~** | NOT | **~5** | **-6** |
|  | **^** | XOR | **5 ^ 1** | **4** |
|  | **<<** | Left shift | **5 << 1** | **10** |
|  | **>>** | Right shift | **5 >> 1** | **2** |
|  | **>>>** | Unsigned right shift | **5 >>> 1** | **2** |
| **String** | **+** | Concatenation | **'Hello' + ' ' + 'World'** | **'Hello World'** |
|  | **+=** | Concatenation assignment | **let greeting = 'Hello'; greeting += ' World';** | **greeting** is **'Hello World'** |
| **Other** | **condition ? expr1 : expr2** | Conditional (ternary) operator | **(5 > 3) ? 'Yes' : 'No'** | **'Yes'** |
|  | **,** | Comma operator | **let a = (1, 2);** | **a** is **2** |
|  | **typeof** | Type of | **typeof 42** | **'number'** |
|  | **instanceof** | Instance of | **let date = new Date(); date instanceof Date** | **true** |

**Operator Precedence in JavaScript**

Operator precedence determines the order in which operations are evaluated in expressions. Operators with higher precedence are evaluated before those with lower precedence. When operators have the same precedence, associativity determines the order of evaluation.

**Examples of Operator Precedence**

Consider the expression:

let result = 3 + 4 \* 2 / (1 - 5) \*\* 2;

To understand how JavaScript evaluates this expression, let's break it down according to operator precedence:

1. **Parentheses ()**: Expressions inside parentheses are evaluated first. (1 - 5) => -4
2. **Exponentiation \*\***: Exponentiation has higher precedence than multiplication, division, addition, and subtraction. (-4) \*\* 2 => 16
3. **Multiplication \* and Division /**: These operators have the same precedence and are evaluated from left to right (left-associative). 4 \* 2 => 8 8 / 16 => 0.5
4. **Addition +**: This operator has lower precedence than multiplication, division, and exponentiation. 3 + 0.5 => 3.5

So, the final result of the expression **3 + 4 \* 2 / (1 - 5) \*\* 2** is **3.5**.

**Operator Precedence Table**

Here's a table summarizing the precedence and associativity of common JavaScript operators:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precedence** | **Operator Type** | **Operators** | **Associativity** | **Description** |
| 1 | Grouping | **()** | n/a | Parentheses |
| 2 | Member Access | **[]**, **.** | left-to-right | Property access |
| 3 | Call/Creation | **new**, **()** | left-to-right | Function call, object creation |
| 4 | Unary | **!**, **typeof**, **+**, **-**, **++**, **--**, **delete**, **void** | right-to-left | Unary operations |
| 5 | Exponentiation | **\*\*** | right-to-left | Exponentiation |
| 6 | Multiplicative | **\***, **/**, **%** | left-to-right | Multiplication, division, modulus |
| 7 | Additive | **+**, **-** | left-to-right | Addition, subtraction |
| 8 | Bitwise Shift | **<<**, **>>**, **>>>** | left-to-right | Bitwise shift operations |
| 9 | Relational | **<**, **<=**, **>**, **>=**, **instanceof**, **in** | left-to-right | Relational comparisons |
| 10 | Equality | **==**, **!=**, **===**, **!==** | left-to-right | Equality comparisons |
| 11 | Bitwise AND | **&** | left-to-right | Bitwise AND |
| 12 | Bitwise XOR | **^** | left-to-right | Bitwise XOR |
| 13 | Bitwise OR | ` | ` | left-to-right |
| 14 | Logical AND | **&&** | left-to-right | Logical AND |
| 15 | Logical OR | ` |  | ` |
| 16 | Conditional (ternary) | **?:** | right-to-left | Ternary operator |
| 17 | Assignment | **=**, **+=**, **-=**, **\*=**, **/=**, **%=** | right-to-left | Assignment operations |
| 18 | Comma | **,** | left-to-right | Comma operator |

**Examples to Illustrate Precedence and Associativity**

1. **Combining Multiplication and Addition:** let result = 2 + 3 \* 4; // 3 \* 4 is evaluated first, result = 2 + 12, so result = 14
2. **Using Parentheses to Change Precedence:** let result = (2 + 3) \* 4; // 2 + 3 is evaluated first, result = 5 \* 4, so result = 20
3. **Combining Multiple Operators:** let result = 10 - 2 \*\* 2 + 3 \* 2; // 2 \*\* 2 is evaluated first, then 10 - 4, then 3 \* 2, then 6 + 6 // result = 10 - 4 + 6, so result = 12

**Template Literals:**

Template literals in JavaScript are a powerful way to work with strings, allowing you to include variables, expressions, and multi-line strings more easily than traditional string concatenation methods. Introduced in ECMAScript 6 (ES6), template literals are enclosed by backticks (**`**) instead of single or double quotes.

**Features of Template Literals**

1. **Variable and Expression Interpolation**:
   * Embed variables and expressions directly within a string.
   * Use the **${}** syntax to insert the value of a variable or the result of an expression.
2. **Multi-line Strings**:
   * Create strings that span multiple lines without the need for concatenation or escape characters for new lines.
3. **Tagged Template Literals**:
   * Process template literals with a function to perform more advanced operations, such as escaping HTML or internationalization.

**Examples**

**1. Variable and Expression Interpolation**

With traditional string concatenation:

let name = "Alice"; let age = 25; let message = "Hello, my name is " + name + " and I am " + age + " years old."; console.log(message); // Output: Hello, my name is Alice and I am 25 years old.

With template literals:

let name = "Alice"; let age = 25; let message = `Hello, my name is ${name} and I am ${age} years old.`; console.log(message); // Output: Hello, my name is Alice and I am 25 years old.

**2. Multi-line Strings**

Without template literals:

let poem = "Roses are red,\n" + "Violets are blue,\n" + "Sugar is sweet,\n" + "And so are you."; console.log(poem);

With template literals:

let poem = `Roses are red,

Violets are blue,

Sugar is sweet,

And so are you.`;

console.log(poem);

**3. Embedding Expressions**

let a = 5; let b = 10;

console.log(`The sum of ${a} and ${b} is ${a + b}.`);

// Output: The sum of 5 and 10 is 15.

**4. Tagged Template Literals**

A tag function can process a template literal. This is useful for escaping, localization, or custom formatting.

function highlight(strings, ...values) { return strings.reduce((result, str, i) => `${result}<b>${str}</b>${values[i] || ''}`, ''); } let name = "Alice"; let age = 25; let message = highlight`Name: ${name}, Age: ${age}`; console.log(message);

// Output: <b>Name: </b>Alice<b>, Age: </b>25

The problems with traditional string concatenation,

* Code becomes verbose and harder to read with multiple **+** operators.
* Embedding complex expressions requires additional parentheses, making the code less readable.
* Creating multi-line strings involves cumbersome concatenation with newline characters (**\n**).
* Increased risk of syntax errors, such as missing **+** operators or mismatched quotes.

**Control Structure:**

Decision making with if/else Control structure

In JavaScript, decision-making is often implemented using **if/else** statements. These statements allow you to execute different blocks of code based on certain conditions. Here’s a basic overview of how to use **if**, **else if**, and **else** in JavaScript.

Syntax (put in a table left right)

if (condition) {

// code to be executed if the condition is true

}

if (condition) {

// code to be executed if the condition is true

} else {

// code to be executed if the condition is false

}

**Basic if Statement**

An **if** statement evaluates a condition (expression). If the condition is true, the block of code inside the **if** statement executes. If the condition is false, the code inside the **if** block does not execute.

let age = 18; if (age >= 18) { console.log("You are an adult."); }

**if/else Statement**

An **if/else** statement allows you to execute one block of code if the condition is true, and another block if the condition is false.

let age = 16;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

**if/else if/else Statement**

An **if/else if/else** statement can be used to test multiple conditions. The first block of code that meets a true condition will execute.

let score = 85;

if (score >= 90) {

console.log("You got an A.");

} else if (score >= 80) {

console.log("You got a B.");

} else if (score >= 70) {

console.log("You got a C.");

} else if (score >= 60) {

console.log("You got a D.");

} else {

console.log("You got an F.");

}

**Nested if Statements**

You can also nest **if** statements within each other to handle more complex conditions.

let age = 20;

let hasPermission = true;

if (age >= 18) {

if (hasPermission) {

console.log("You can enter the club.");

} else {

console.log("You need permission to enter the club.");

}

} else {

console.log("You are not allowed to enter the club.");

}

**Type Conversion and Type Coercion**

<https://www.almabetter.com/bytes/tutorials/javascript/coercion-and-type-conversion-in-javascript>