JavaScript Operators

JavaScript Operators

Operators are the backbone of expressions in JavaScript. They interact with values through arithmetic, assignment, comparison, logical flow, property inspection, and more. Because JavaScript is dynamically typed and uses automatic type coercion in many places, understanding not just “what it does” but also “how it decides” is crucial to avoiding surprises.

Arithmetic operators

Arithmetic works on numbers, with special cases for strings and BigInt. JavaScript Numbers use IEEE 754 double-precision, which introduces rounding behavior for certain decimals.

- **Addition (+): Adds numbers or concatenates strings**
    - If either operand is a string, the operation becomes string concatenation. The order of operations matters because evaluation goes left to right.
- 5 + 3;   // 8
- "Hello " + "JS";   // "Hello JS"
- 1 + "2";   // "12" (number → string, then concatenation)
- "1" + 2 + 3;   // "123" (left operand is string, chain concatenates)
- 1 + 2 + "3";   // "33" (numbers add first, then concatenation)
    - **Pitfall:** Number + String yields String; be explicit if you need numeric addition (e.g., Number("2")).
- **Subtraction (-), multiplication (*), division (/), remainder (%), exponentiation (**): Numeric operations with coercion to number where possible**
    - 10 - 4;   // 6
    - "10" - 4;   // 6 (string → number)
    - "a" - 1;   // NaN

    - 4 * 2;   // 8
    - "6" * "3";   // 18 (both coerced)

    - 8 / 2;   // 4

JavaScript Operators

- 1 / 0;   // Infinity
- 0 / 0;   // NaN

- 10 % 3;   // 1
- -10 % 3;   // -1 (sign follows dividend)

- 2 ** 3;   // 8
- 9 ** 0.5;   // 3
  // Precedence with unary negation:
- -(2 ** 3);   // -8
- // -2 ** 3 is a SyntaxError in strict mode; parenthesize the base.

Increment/decrement (++/--): Mutate a variable by 1; prefix returns the updated value, postfix returns the old value

  - let x = 5;
  - ++x; // 6 (x is 6; expression value is 6)
  - x++; // 6 (expression returns old value 6; x becomes 7)
  - --x; // 6 (x is 6; expression value is 6)
- **Precision tips:**
  - **Floating-point:** 0.1 + 0.2 !== 0.3. Prefer tolerance checks or integer scaling.
  - Math.abs((0.1 + 0.2) - 0.3) < 1e-10; // true
  - // or scale:
  - (1 + 2) / 10 === 0.3; // true
  - **BigInt:** Use BigInt for integers beyond 2^53 − 1 to avoid precision loss.

Assignment operators

Assignments set or update variables. Some combine an operation with assignment, and logical forms short-circuit.

- **Simple assignment (=): Set a variable**
  - let a = 10;   // const cannot be reassigned; let and var can.
- **Compound arithmetic (+=, -=, *=, /=, %=, =): Update in place**
  - let n = 10;

JavaScript Operators

  - n += 5;   // 15
  - n *= 2;   // 30
  - n **= 2;   // 900
- **Bitwise compound (&=, |=, ^=, <<=, >>=, >>>=): Combine with 32-bit ops**
  - let m = 6; // 110
  - m &= 3;   // 010 -> 2
- **Logical assignment (||=, &&=, ??=): Short-circuit semantics applied before assignment**
  - let title = "";
  - title ||= "Untitled"; // "" is falsy -> "Untitled"

  - let enabled = true;
  - enabled &&= false;   // true is truthy -> sets to false

  - let count = null;
  - count ??= 0;   // nullish -> sets to 0
    - **Use ||= for "fallback when falsy"** (treats 0, "", NaN as empty).
    - **Use ??= for "fallback when nullish"** (only null/undefined).
- **Destructuring assignment: Extract from arrays/objects**
  - const [first, second] = [10, 20];
  - const { name, age } = { name: "Aman", age: 22 };
- **Notes:**
  - **Right associativity:** a = b = 5 sets both to 5 (b = 5 happens first).
  - **const bindings:** You can mutate the contents of an object bound to const, but not reassign the binding itself.

Comparison and relational operators

Comparisons return booleans and drive conditionals. Prefer strict equality to avoid coercion surprises.

- **Strict equality/inequality (=, !): Check value and type**
  - 5 === 5;   // true
  - "5" === 5;   // false
  - NaN === NaN;   // false (use Number.isNaN)
  - Number.isNaN(NaN); // true

JavaScript Operators

- **Loose equality/inequality (==, !=): Allow type coercion**
    - "5" == 5;   // true (string -> number)
    - true == 1;   // true
    - null == undefined;   // true
    - 0 == "";   // true
        - **Best practice:** Use =/! unless you explicitly want coercion (rare).
- **Relational (<, <=, >, >=): Numeric for numbers; lexicographic for strings**
    - 10 > 5;   // true
    - "10" < "9";   // true ("1" < "9" at first character)
    - "apple" < "banana";   // true (lexicographic)
    - ({}) > 1;   // false (object -> NaN)
- **Object reference equality:** Only the same reference is strictly equal
    - [] === [];   // false
    - const arr = [];
    - arr === arr;   // true
- **BigInt comparisons:**
    - 10n > 9n;   // true
    - 1n === 1;   // false (different types)
    - 1n == 1;   // true (loose equality may coerce)

Logical and nullish operators

These work on “truthy” and “falsy” values with short-circuit behavior. They return one of the original operands, not necessarily a boolean.

- **AND (&&): Returns the first falsy operand, or the last if all are truthy**
    - true && "ok";   // "ok"
    - 0 && "nope";   // 0
- **OR (||): Returns the first truthy operand, or the last if all are falsy**
    - "" || "default"; // "default"
    - "hi" || "default"; // "hi"
- **NOT (!): Coerces to boolean and negates; !!x is a common “toBoolean” pattern**
    - !0;   // true
    - !!"text";   // true
- **Nullish coalescing (??): Only falls back when left is null or undefined**

  - 0 ?? 5;   // 0
  - null ?? "N/A";   // "N/A"
- **Truthiness reminder:** falsy values are false, 0, -0, 0n, "", null, undefined, NaN. Everything else is truthy (including "0", [], {}).
- **Mixing precedence caution:** ?? cannot be mixed with || or && without parentheses in some environments. Use grouping to be explicit.
  - a ?? (b || c);
  - (a && b) ?? c;

Bitwise and shift operators

Operate on 32-bit signed integers (Numbers are coerced via ToInt32). For BigInt, both operands must be BigInt.

- **Basic bitwise (&, |, ^, ~): AND, OR, XOR, NOT**
  - 5 & 1; // 0101 & 0001 = 0001 => 1
  - 5 | 2; // 0101 | 0010 = 0111 => 7
  - 5 ^ 1; // 0101 ^ 0001 = 0100 => 4
  - ~5;   // bitwise NOT => -(5 + 1) = -6
- **Shifts (<<, >>, >>>): Left, signed right, unsigned right**
  - 5 << 1;   // 10
  - -8 >> 1;   // -4 (arithmetic shift preserves sign bit)
  - -8 >>> 1;   // large positive (logical shift fills with zeros)
- **Notes:**
  - **Numbers → 32-bit:** Fractions are truncated; values wrap at 32-bit.
  - **BigInt bitwise works only with BigInt:** Mixing Number and BigInt throws TypeError.
  - 1n << 3n; // 8n

Ternary and comma operators

Use for concise expressions, but keep readability in mind.

- **Conditional (?:): Inline if/else**
  - const age = 18;
  - const status = age >= 18 ? "Adult" : "Minor";
    - **Tip:** Keep branches short; prefer if/else when nesting.

JavaScript Operators

- **Comma (,): Evaluate left to right, return last value**
    - let a, b;
    - const result = (a = 1, b = 2, a + b); // 3
        - **Use sparingly:** It can obscure intent outside of for-loops.

Unary operators

Single-operand operators for numeric conversion, type inspection, property deletion, and control.

- **Unary plus/minus (+, -): Coerce to number and optionally negate**
    - +"42";   // 42
    - +true;   // 1
    - +"abc";   // NaN
    - -"5";   // -5
- **Logical NOT (!) and bitwise NOT (~):**
    - !value;   // boolean negation after truthiness coercion
    - ~5;   // -6 (equals -(5 + 1))
- **typeof: Returns a type string (with historical quirks)**
    - typeof 5;   // "number"
    - typeof "x";   // "string"
    - typeof null;   // "object" (legacy quirk)
    - typeof undefined;   // "undefined"
    - typeof [];   // "object"
    - typeof (() => {});   // "function"
- **delete: Removes a property from objects (not variables)**
    - const obj = { a: 1 };
    - delete obj.a;   // true; obj.a is now undefined
    - // Cannot delete local let/const bindings.
- **void: Evaluate an expression and return undefined**
    - void 0;   // undefined
    - void (someCall()); // runs someCall(), expression value is undefined
- **await: Pause inside async functions until a Promise settles**
    - const data = await fetch("/api").then(r => r.json());
- **Increment/Decrement (++/--):** See arithmetic notes for prefix vs postfix behavior.

JavaScript Operators

Property and type relation operators

Inspect presence of keys, prototype relations, and safely access properties.

- **in: Check if a property key exists (own or inherited)**
  - const obj = { length: 10 };
  - "length" in obj;   // true
  - "toString" in obj;   // true (inherited)
- **instanceof: Check prototype chain for constructor**
  - [] instanceof Array;   // true
  - new Date() instanceof Date;   // true
    - **Cross-realm caution:** Objects from different iframes/windows may fail instanceof. Prefer Array.isArray() for arrays.
- **Optional chaining (?.): Safely access nested properties/methods**
  - const user = { address: { city: "Calamba" } };
  - user.address?.city;   // "Calamba"
  - user.contact?.phone;   // undefined (no error)
  - user.getName?.();   // calls if function exists; else undefined
  - user.prefs?.themes?.[0];   // safe array access

- const city = user.address?.city ?? "Unknown"; // combine with ??
  - **Stops only on null/undefined:** Values like 0 or "" don’t stop the chain.

BigInt operators

BigInt represents whole numbers of arbitrary size. Do not mix with Number in most operations.

- **Arithmetic and comparison:**
  - 10n + 20n;   // 30n
  - 7n / 2n;   // 3n (truncates toward zero)
  - 2n ** 53n;   // huge integer
  - 100n > 99n; // true

  - 1n === 1;   // false (different types)
- **Bitwise with BigInt:** Supported only when both operands are BigInt
  - 1n << 60n;   // 1152921504606846976n
- **Interoperability:**
  - **Explicit conversion:** Number(1n) or BigInt(1), but be mindful of precision limits when converting large BigInts to Number.

String operators and concatenation

Strings use + and += for concatenation. Be mindful of coercion when mixing with numbers.

- **Concatenation and chaining:**
  - "Hello" + " " + "World"; // "Hello World"
  - let s = "Hi";
  - s += " there";   // "Hi there"
  - "Result: " + 2 + 3;   // "Result: 23"
  - 2 + 3 + " is five";   // "5 is five"
- **Template literals: Prefer for readability and expressions**
  - const name = "CodeFrill";
  - `Hello, ${name}!`;   // "Hello, CodeFrill!"
- **Tip:** If you must add numbers then concatenate, force numeric addition first or use parentheses/template literals.

Precedence and associativity essentials

Operator precedence determines grouping; associativity determines tie-breaking direction.

- **Key rules:**
  - **Exponentiation (**) binds tighter than unary negation:** Write -(a ** b).
  - *Multiplicative (/ %) > additive (+ -).**
  - Comparisons (< > <= >=) occur before logical AND (&&) and OR (||).

    - Nullish (??) has lower precedence than && and ||, and cannot be mixed with them without parentheses in some environments.
    - Assignment (=, +=, …) is right-associative: a = b = 5.
- **Clarity tip:** When in doubt, add parentheses. They document intent and prevent surprises.

Common pitfalls and best practices

- **Prefer strict equality:** Use =/! to avoid unintended coercion with ==/!=.
- **Choose the right fallback operator:**
    - **|| for "falsy fallback"** (treats 0/""/NaN as empty).
    - **?? for "nullish fallback"** (keeps 0/"" as valid).
- **Avoid deep ternary chains:** Use if/else for readability.
- **Mind + with strings and numbers:** If concatenation isn't intended, coerce explicitly or use template literals.
- **Floating-point quirks:** Avoid direct equality checks on decimals; use a tolerance or rational scaling.
- **Object equality is by reference:** Two identical object literals are not equal unless they are the same reference.
- **Deletion scope:** delete removes object properties, not variables declared with let/const.
- **Bitwise on Numbers truncates to 32-bit:** Don't use bitwise for large integer math with Numbers—use BigInt if needed.
- **Optional chaining only stops on null/undefined:** Don't expect it to stop on falsy values like 0 or "".