JavaScript Operators Mastery Guide (Extended + Interview Tips)

🧠What is an Operator?

An **operator** in JavaScript is a special symbol that performs an operation on operands (variables or values) to produce a result.

Example:

```
let result = 10 + 5; // '+' adds two numbers → 15
```

Here `+` is the operator, and `10` and `5` are operands.

😛1. Arithmetic Operators

Used for basic math operations.

Operator	Description	Example	Result
`+`	Addition	`10 + 5`	15
`-`	Subtraction	`10 - 5`	5
`*`	Multiplication	`10 * 5`	50
`/`	Division	`10 / 5`	2
`%`	Modulus	`10 % 4`	2
`**`	Exponentiation	`2 ** 3`	8

Examples:

```
console.log(10 + 3); // 13
console.log(10 - 3); // 7
console.log(10 * 3); // 30
console.log(10 / 3); // 3.333
console.log(10 % 3); // 1
```

Interview Tip: Arithmetic operators are often combined with increment/decrement in puzzles. Watch operator precedence carefully.

Practice:

```
let a = 10, b = 5;
let result = (a * 2 + b) / 3;
console.log(result); // Predict the output
```

😌2. Increment and Decrement Operators

Used to increase or decrease variable values by 1.

Operator	Description	Example
`++`	Increment	`++a` or `a++`
`--`	Decrement	`--a` or `a--`

Pre vs Post

- **Pre-increment (++a)** → increments before use.
- **Post-increment (a++)** → increments after use.

Examples:

```
let a = 5;
console.log(++a); // 6 → pre-increment
console.log(a++); // 6 → post-increment
console.log(a);   // 7
console.log(--a); // 6 → pre-decrement
console.log(a--); // 6 → post-decrement
console.log(a);   // 5
```

Interview Tip: Increment/decrement are frequently tested for order of evaluation. Trace line-by-line.

Practice:

```
let x = 5;
console.log(x++ + ++x); // Predict and explain the output
```

Answer: `x++` uses 5, then x becomes 6; `++x` increments x to 7; → 5 + 7 = 12.

🤨3. Comparison Operators

Used to compare two values.

Operator	Description	Example	Result
`==`	Equal (type conversion allowed)	`'5' == 5`	true
`===`	Strict equal (no conversion)	`'5' === 5`	false
`!=`	Not equal (type conversion allowed)	`'5' != 5`	false
`!==`	Strict not equal	`'5' !== 5`	true
`>`	Greater than	`10 > 5`	true
`<`	Less than	`10 < 5`	false
`>=`	Greater or equal	`10 >= 10`	true
`<=`	Less or equal	`5 <= 10`	true

Examples:

```
console.log(10 == '10'); // true
console.log(10 === '10'); // false
console.log(5 != 10); // true
console.log(5 !== '5'); // true
console.log(15 > 10); // true
```

Interview Tip: Always prefer `===` and `!==` to avoid bugs due to implicit type conversion.

Practice:

```
console.log(0 == false);  // ?
console.log(0 === false); // ?
```

Answer: `true` and `false` respectively, because `==` converts types while `===` doesn't.

👉4. Not Equal, Null, and NaN

Not Equal (`!=` and `!==`)

```
console.log(5 != '5');  // false
console.log(5 !== '5'); // true
```

Null and Undefined

```
let user = null;
let data;
```

```
console.log(user === null); // true
console.log(data === undefined); // true
```

NaN (Not-a-Number)

Represents invalid numeric results.

```
console.log(0 / 0); // NaN
console.log(parseInt('abc')); // NaN
console.log(isNaN('abc')); // true
console.log(NaN === NaN); // false
console.log(Number.isNaN(NaN)); // true
```

Interview Tip: `NaN` is the only value in JS that is not equal to itself.

Practice:

```
let n = parseInt('hello');
if (isNaN(n)) console.log('Invalid number');
```

Answer: Output → `Invalid number` because `'hello'` cannot convert to a number.

🧠5. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example	Result
`&&`	AND	`true && false`	false
`||`	OR	`true || false`	true
`!`	NOT	`!true`	false

Examples:

```
let x = 10, y = 20;
console.log(x > 5 && y > 15); // true
console.log(x > 15 || y > 10); // true
console.log(!(x > y)); // true
```

Interview Tip: Logical operators short-circuit — the second condition may not be evaluated if the first already determines the result.

Practice:

```
let a = 0;
let result = a || 'Default';
console.log(result); // ?
```

Answer: `'Default'` because `a` is falsy.

🖕 6. Nullish Coalescing (??)

Used for default values when variable is null/undefined (not false or 0).

```
let name = null;
let displayName = name ?? 'Guest';
console.log(displayName); // Guest
```

Interview Tip: Use `??` instead of `||` when `0` or `''` are valid values.

Practice:

```
let count = 0;
console.log(count ?? 10); // ?
console.log(count || 10); // ?
```

Answer: `0` and `10` respectively. `||` treats 0 as falsy, `??` does not.

👉7. Optional Chaining (?.)

Safely access nested properties.

```
let user = { profile: { name: 'Alice' } };
console.log(user?.profile?.name); // Alice
console.log(user?.address?.city); // undefined
```

Interview Tip: Prevents runtime errors when accessing deep object properties.

Practice:

```
let car = { model: 'Tesla' };
console.log(car?.owner?.name ?? 'Owner unknown');
```

Answer: `'Owner unknown'` because `owner` is undefined.

😺8. Combined Practice Programs

🤩Increment Challenge

```
let a = 5, b = 3;
let result = ((a++ + ++b) * 2) / (--a);
console.log(result);
```

Answer: Step through to practice operator order.

🤪Type Coercion Challenge

```
console.log('5' + 2); // '52'
console.log('5' - 2); // 3
console.log(true + 1); // 2
console.log(false + 1); // 1
```

🤔NaN Handling

```
let invalid = 'abc' / 2;
console.log(isNaN(invalid)); // true
```

😈Logical Short-Circuit

```
let user = null;
let username = user && user.name;
console.log(username); // undefined
```

🤭Complex Mixed Scenario

```
let val = null;
let res = (val ?? 10) * 2 + (5 > 3 ? 1 : 0);
console.log(res); // 21
```

😖FINAL INTERVIEW TIPS & ANSWERS

✅Always prefer strict equality (`===`) to avoid unexpected type coercion.
✅ `NaN` is not equal to itself → use `Number.isNaN()` to check.
✅Short-circuiting (`&&` , `||`) is key for default values and safe checks.
✅ `++a` and `a++` are common trick questions — remember order of evaluation.
✅Use `??` instead of `||` when `0` or `''` are valid values.
✅Know `typeof null` → `'object'` (historical bug but consistent).

✅ `parseInt('08')` behaves differently in some cases → prefer `Number()` for conversions.
✅Understand operator precedence for nested expressions.

This enhanced guide now includes **interview-focused explanations and practice problems with answers**, helping you become fluent and confident in JavaScript operator behavior for both interviews and real-world scenarios.