mdn web docs

Operator precedence

Operator precedence determines how operators are parsed concerning each other. Operators with higher precedence become the operands of operators with lower precedence.

Try it

JavaScript Demo: Expressions - Operator precedence

```
1  console.log(3 + 4 * 5); // 3 + 20
2  // Expected output: 23
3
4  console.log(4 * 3 ** 2); // 4 * 9
5  // Expected output: 36
6
7  let a;
8  let b;
9
10 console.log(a = b = 5);
11 // Expected output: 5
12
```

Run ›

Reset

Precedence And Associativity

Consider an expression describable by the representation below, where both `OP1` and `OP2` are fill-in-the-blanks for OPerators.

```
a OP1 b OP2 c
```

The combination above has two possible interpretations:

```
(a OP1 b) OP2 c
a OP1 (b OP2 c)
```

Which one the language decides to adopt depends on the identity of `OP1` and `OP2`.

If `OP1` and `OP2` have different precedence levels (see the table below), the operator with the higher *precedence* goes first and associativity does not matter. Observe how multiplication has higher precedence than addition and executed first, even though addition is written first in the code.

```
console.log(3 + 10 * 2); // 23
console.log(3 + (10 * 2)); // 23, because parentheses here are superfluous
console.log((3 + 10) * 2); // 26, because the parentheses change the order
```

Within operators of the same precedence, the language groups them by *associativity*. *Left-associativity* (left-to-right) means that it is interpreted as `(a OP1 b) OP2 c`, while *right-associativity* (right-to-left) means it is interpreted as `a OP1 (b OP2 c)`. Assignment operators are right-associative, so you can write:

```
a = b = 5; // same as writing a = (b = 5);
```

with the expected result that `a` and `b` get the value 5. This is because the assignment operator returns the value that is assigned. First, `b` is set to 5. Then the `a` is also set to 5 — the return value of `b = 5`, a.k.a. right operand of the assignment.

As another example, the unique exponentiation operator has right-associativity, whereas other arithmetic operators have left-associativity.

```
const a = 4 ** 3 ** 2; // Same as 4 ** (3 ** 2); evaluates to 262144
const b = 4 / 3 / 2; // Same as (4 / 3) / 2; evaluates to 0.6666...
```

Operators are first grouped by precedence, and then, for adjacent operators that have the same precedence, by associativity. So, when mixing division and exponentiation, the exponentiation always comes before the division. For example, `2 ** 3 / 3 ** 2` results in 0.8888888888888888 because it is the same as `(2 ** 3) / (3 ** 2)`.

For prefix unary operators, suppose we have the following pattern:

```
OP1 a OP2 b
```

where `OP1` is a prefix unary operator and `OP2` is a binary operator. If `OP1` has higher precedence than `OP2`, then it would be grouped as `(OP1 a) OP2 b`; otherwise, it would be `OP1 (a OP2 b)`.

```
const a = 1;
const b = 2;
typeof a + b; // Equivalent to (typeof a) + b; result is "number2"
```

If the unary operator is on the second operand:

```
a OP2 OP1 b
```

Then the binary operator `OP2` must have lower precedence than the unary operator `OP1` for it to be grouped as `a OP2 (OP1 b)`. For example, the following is invalid:

```
function* foo() {
  a + yield 1;
}
```

Because `+` has higher precedence than `yield`, this would become `(a + yield) 1` — but because `yield` is a reserved word in generator functions, this would be a syntax error. Luckily, most unary operators have higher precedence than binary operators and do not suffer from this pitfall.

If we have two prefix unary operators:

```
OP1 OP2 a
```

Then the unary operator closer to the operand, `OP2`, must have higher precedence than `OP1` for it to be grouped as `OP1 (OP2 a)`. It's possible to get it the other way and end up with `(OP1 OP2) a`:

```
async function* foo() {
  await yield 1;
}
```

Because `await` has higher precedence than `yield`, this would become `(await yield) 1`, which is awaiting an identifier called `yield`, and a syntax error. Similarly, if you have `new !A;`, because `!` has lower precedence than `new`, this would become `(new !) A`, which is obviously invalid. (This code looks nonsensical to write anyway, since `!A` always produces a boolean, not a constructor function.)

For postfix unary operators (namely, `++` and `--`), the same rules apply. Luckily, both operators have higher precedence than any binary operator, so the grouping is always what you would expect. Moreover, because `++` evaluates to a *value*, not a *reference*, you can't chain multiple increments together either, as you may do in C.

```
let a = 1;
a++++; // SyntaxError: Invalid left-hand side in postfix operation.
```

Operator precedence will be handled *recursively*. For example, consider this expression:

```
1 + 2 ** 3 * 4 / 5 >> 6
```

First, we group operators with different precedence by decreasing levels of precedence.

1. The `**` operator has the highest precedence, so it's grouped first.
2. Looking around the `**` expression, it has `*` on the right and `+` on the left. `*` has higher precedence, so it's grouped first. `*` and `/` have the same precedence, so we group them together for now.
3. Looking around the `*`/`/` expression grouped in 2, because `+` has higher precedence than `>>`, the former is grouped.

```
   (1 + ( (2 ** 3) * 4 / 5) ) >> 6
// |    | └─ 1. ─┘        | |
// |    └─────── 2. ──────┘ |
// └────────── 3. ──────────┘
```

Within the `*`/`/` group, because they are both left-associative, the left operand would be grouped.

```
   (1 + ( ( (2 ** 3) * 4 ) / 5) ) >> 6
// |    | | └─ 1. ─┘     |    | |
// |    └─|─────── 2. ──|────┘ |
// └──────|──── 3. ─────|──────┘
//        └──── 4. ─────┘
```

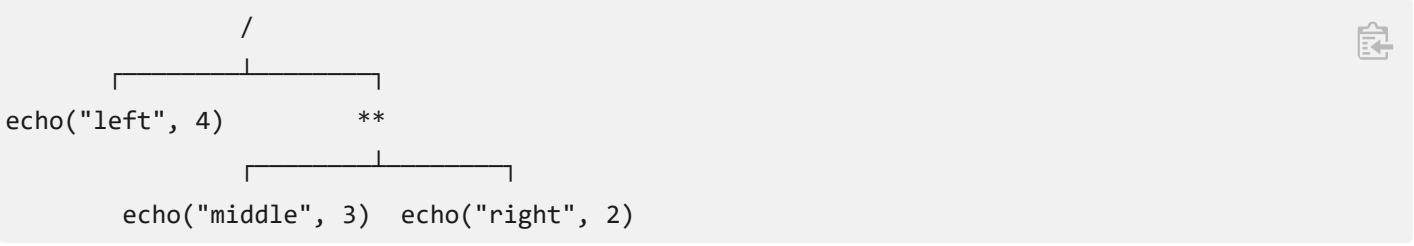
Note that operator precedence and associativity only affect the order of evaluation of *operators* (the implicit grouping), but not the order of evaluation of *operands*. The operands are always evaluated from left-to-right. The higher-precedence expressions are always evaluated first, and their results are then composed according to the order of operator precedence.

```
function echo(name, num) {
  console.log(`Evaluating the ${name} side`);
  return num;
```

```
}
// Exponentiation operator (**) is right-associative,
// but all call expressions (echo()), which have higher precedence,
// will be evaluated before ** does
console.log(echo("left", 4) ** echo("middle", 3) ** echo("right", 2));
// Evaluating the left side
// Evaluating the middle side
// Evaluating the right side
// 262144

// Exponentiation operator (**) has higher precedence than division (/),
// but evaluation always starts with the left operand
console.log(echo("left", 4) / echo("middle", 3) ** echo("right", 2));
// Evaluating the left side
// Evaluating the middle side
// Evaluating the right side
// 0.4444444444444444
```

If you are familiar with binary trees, think about it as a post-order traversal.



After all operators have been properly grouped, the binary operators would form a binary tree. Evaluation starts from the outermost group — which is the operator with the lowest precedence (`/` in this case). The left operand of this operator is first evaluated, which may be composed of higher-precedence operators (such as a call expression `echo("left", 4)`). After the left operand has been evaluated, the right operand is evaluated in the same fashion. Therefore, all leaf nodes — the `echo()` calls — would be visited left-to-right, regardless of the precedence of operators joining them.

Short-circuiting

In the previous section, we said "the higher-precedence expressions are always evaluated first" — this is generally true, but it has to be amended with the acknowledgement of *short-*

circuiting, in which case an operand may not be evaluated at all.

Short-circuiting is jargon for conditional evaluation. For example, in the expression `a && (b + c)`, if `a` is falsy, then the sub-expression `(b + c)` will not even get evaluated, even if it is grouped and therefore has higher precedence than `&&`. We could say that the logical AND operator (`&&`) is "short-circuited". Along with logical AND, other short-circuited operators include logical OR (`||`), nullish coalescing (`??`), and optional chaining (`?.`).

```
a || (b * c); // evaluate `a` first, then produce `a` if `a` is "truthy"
a && (b < c); // evaluate `a` first, then produce `a` if `a` is "falsy"
a ?? (b || c); // evaluate `a` first, then produce `a` if `a` is not `null` and not `undefined`
a?.b.c; // evaluate `a` first, then produce `undefined` if `a` is `null` or `undefined`
```

When evaluating a short-circuited operator, the left operand is always evaluated. The right operand will only be evaluated if the left operand cannot determine the result of the operation.

> **Note:** The behavior of short-circuiting is baked in these operators. Other operators would *always* evaluate both operands, regardless if that's actually useful — for example, `NaN * foo()` will always call `foo`, even when the result would never be something other than `NaN`.

The previous model of a post-order traversal still stands. However, after the left subtree of a short-circuiting operator has been visited, the language will decide if the right operand needs to be evaluated. If not (for example, because the left operand of `||` is already truthy), the result is directly returned without visiting the right subtree.

Consider this case:

```
function A() { console.log('called A'); return false; }
function B() { console.log('called B'); return false; }
function C() { console.log('called C'); return true; }

console.log(C() || B() && A());
```

```
// Logs:
// called C
// true
```

Only `C()` is evaluated, despite `&&` having higher precedence. This does not mean that `||` has higher precedence in this case — it's exactly *because* `(B() && A())` has higher precedence that causes it to be neglected as a whole. If it's re-arranged as:

```
console.log(A() && C() || B());
// Logs:
// called A
// called B
// false
```

Then the short-circuiting effect of `&&` would only prevent `C()` from being evaluated, but because `A() && C()` as a whole is `false`, `B()` would still be evaluated.

However, note that short-circuiting does not change the final evaluation outcome. It only affects the evaluation of *operands*, not how *operators* are grouped — if evaluation of operands doesn't have side effects (for example, logging to the console, assigning to variables, throwing an error), short-circuiting would not be observable at all.

The assignment counterparts of these operators (`&&=`, `||=`, `??=`) are short-circuited as well. They are short-circuited in a way that the assignment does not happen at all.

Table

The following table lists operators in order from highest precedence (18) to lowest precedence (1).

Several notes about the table:

1. Not all syntax included here are "operators" in the strict sense. For example, spread `...` and arrow `=>` are typically not regarded as operators. However, we still included them to

show how tightly they bind compared to other operators/expressions.

2. The left operand of an exponentiation `**` (precedence 13) cannot be one of the unary operators with precedence 14 without grouping, or there will be a `SyntaxError`. That means, although `-1 ** 2` is technically unambiguous, the language requires you to use `(-1) ** 2` instead.
3. The operands of nullish coalescing `??` (precedence 3) cannot be a logical OR `||` (precedence 3) or logical AND `&&` (precedence 4). That means you have to write `(a ?? b) || c` or `a ?? (b || c)`, instead of `a ?? b || c`.
4. Some operators have certain operands that require expressions narrower than those produced by higher-precedence operators. For example, the right-hand side of member access `.` (precedence 17) must be an identifier instead of a grouped expression. The left-hand side of arrow `=>` (precedence 2) must be an arguments list or a single identifier instead of some random expression.
5. Some operators have certain operands that accept expressions wider than those produced by higher-precedence operators. For example, the bracket-enclosed expression of bracket notation `[ … ]` (precedence 17) can be any expression, even comma (precedence 1) joined ones. These operators act as if that operand is "automatically grouped". In this case we will omit the associativity.

Precedence	Operator type	Associativity	Individual operators
18	Grouping	n/a	`( … )`
17	Member Access	left-to-right	`… . …`
	Optional chaining		`… ?. …`
	Computed Member Access	n/a	`… [ … ]`
	`new` (with argument list)		`new … ( … )`
	Function Call		`… ( … )`

16	`new` (without argument list)	n/a	`new …`
15	Postfix Increment	n/a	`… ++`
	Postfix Decrement		`… --`
14	Logical NOT (!)	n/a	`! …`
	Bitwise NOT (~)		`~ …`
	Unary plus (+)		`+ …`
	Unary negation (-)		`- …`
	Prefix Increment		`++ …`
	Prefix Decrement		`-- …`
	`typeof`		`typeof …`
	`void`		`void …`
	`delete`		`delete …`
	`await`		`await …`
13	Exponentiation (**)	right-to-left	`… ** …`
12	Multiplication (*)	left-to-right	`… * …`
	Division (/)		`… / …`
	Remainder (%)		`… % …`
11	Addition (+)	left-to-right	`… + …`
	Subtraction (-)		`… - …`
10	Bitwise Left Shift (<<)	left-to-right	`… << …`
	Bitwise Right Shift (>>)		`… >> …`

	Bitwise Unsigned Right Shift (>>>)		`… >>> …`
9	Less Than (<)	left-to-right	`… < …`
	Less Than Or Equal (<=)		`… <= …`
	Greater Than (>)		`… > …`
	Greater Than Or Equal (>=)		`… >= …`
	`in`		`… in …`
	`instanceof`		`… instanceof …`
8	Equality (==)	left-to-right	`… == …`
	Inequality (!=)		`… != …`
	Strict Equality (===)		`… === …`
	Strict Inequality (!==)		`… !== …`
7	Bitwise AND (&)	left-to-right	`… & …`
6	Bitwise XOR (^)	left-to-right	`… ^ …`
5	Bitwise OR (|)	left-to-right	`… | …`
4	Logical AND (&&)	left-to-right	`… && …`
3	Logical OR (||)	left-to-right	`… || …`
	Nullish coalescing operator (??)		`… ?? …`
2	Assignment	right-to-left	`… = …`
			`… += …`

			`… -= …`
			`… **= …`
			`… *= …`
			`… /= …`
			`… %= …`
			`… <<= …`
			`… >>= …`
			`… >>>= …`
			`… &= …`
			`… ^= …`
			`… |= …`
			`… &&= …`
			`… ||= …`
			`… ??= …`
	Conditional (ternary) operator	right-to-left (Groups on expressions after `?`)	`… ? … : …`
	Arrow (=>)	right-to-left	`… => …`
	`yield`	n/a	`yield …`
	`yield*`		`yield* …`
	Spread (...)		`... …`
1	Comma / Sequence	left-to-right	`… , …`

This page was last modified on Apr 5, 2023 by MDN contributors.