Expressions and operators

This chapter describes JavaScript's expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, ternary and more.

At a high level, an *expression* is a valid unit of code that resolves to a value. There are two types of expressions: those that have side effects (such as assigning values) and those that purely *evaluate*.

The expression `x = 7` is an example of the first type. This expression uses the `=` *operator* to assign the value seven to the variable `x`. The expression itself evaluates to `7`.

The expression `3 + 4` is an example of the second type. This expression uses the `+` operator to add `3` and `4` together and produces a value, `7`. However, if it's not eventually part of a bigger construct (for example, a variable declaration like `const z = 3 + 4`), its result will be immediately discarded — this is usually a programmer mistake because the evaluation doesn't produce any effects.

JavaScript modules

▶ Intermediate

▶ Advanced

References:

▶ Built-in objects

▶ Expressions & operators

▶ Statements & declarations

▶ Functions

▶ Classes

▶ Errors

▶ Misc

As the examples above also illustrate, all complex expressions are joined by *operators*, such as `=` and `+`. In this section, we will introduce the following operators:

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- BigInt operators
- String operators
- Conditional (ternary) operator
- Comma operator
- Unary operators
- Relational operators

These operators join operands either formed by higher-precedence operators or one of the basic expressions. A complete and detailed list of operators and expressions is also available in the reference.

The *precedence* of operators determines the order they are applied when evaluating an expression. For example:

```
const x = 1 + 2 * 3;
const y = 2 * 3 + 1;
```

Despite `*` and `+` coming in different orders, both expressions would result in `7` because `*` has precedence over `+`, so the `*`-joined

expression will always be evaluated first. You can override operator precedence by using parentheses (which creates a grouped expression — the basic expression). To see a complete table of operator precedence as well as various caveats, see the Operator Precedence Reference page.

JavaScript has both *binary* and *unary* operators, and one special ternary operator, the conditional operator. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example, `3 + 4` or `x * y`. This form is called an *infix* binary operator, because the operator is placed between two operands. All binary operators in JavaScript are infix.

A unary operator requires a single operand, either before or after the operator:

```
operator operand
operand operator
```

For example, `x++` or `++x`. The `operator operand` form is called a *prefix* unary operator, and the `operand operator` form is called a *postfix* unary operator. `++` and `--` are the only postfix operators in JavaScript — all other operators, like `!`, `typeof`, etc. are prefix.

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (`=`), which assigns the value of its right operand to its left operand. That is, `x = f()` is an assignment expression that assigns the value of `f()` to `x`.

There are also compound assignment operators that are shorthand for the operations listed in the following table:

Name	Shorthand operator	Meaning
Assignment	`x = f()`	`x = f()`
Addition assignment	`x += f()`	`x = x + f()`
Subtraction assignment	`x -= f()`	`x = x - f()`
Multiplication assignment	`x *= f()`	`x = x * f()`
Division assignment	`x /= f()`	`x = x / f()`
Remainder assignment	`x %= f()`	`x = x % f()`
Exponentiation assignment	`x **= f()`	`x = x ** f()`
Left shift assignment	`x <<= f()`	`x = x << f()`
Right shift	`x >>= f()`	`x = x >>`

Name	Shorthand operator	Meaning
assignment		`f()`
Unsigned right shift assignment	`x >>>= f()`	`x = x >>> f()`
Bitwise AND assignment	`x &= f()`	`x = x & f()`
Bitwise XOR assignment	`x ^= f()`	`x = x ^ f()`
Bitwise OR assignment	`x |= f()`	`x = x | f()`
Logical AND assignment	`x &&= f()`	`x && (x = f())`
Logical OR assignment	`x ||= f()`	`x || (x = f())`
Nullish coalescing assignment	`x ??= f()`	`x ?? (x = f())`

Assigning to properties

If an expression evaluates to an object, then the left-hand side of an assignment expression may make assignments to properties of that expression. For example:

```
const obj = {};

obj.x = 3;
console.log(obj.x); // Prints 3.
console.log(obj); // Prints { x: 3 }.
```

```
const key = "y";
obj[key] = 5;
console.log(obj[key]); // Prints 5.
console.log(obj); // Prints { x: 3, y: 5 }.
```

For more information about objects, read Working with Objects.

If an expression does not evaluate to an object, then assignments to properties of that expression do not assign:

```
const val = 0;
val.x = 3;

console.log(val.x); // Prints undefined.
console.log(val); // Prints 0.
```

In strict mode, the code above throws, because one cannot assign properties to primitives.

It is an error to assign values to unmodifiable properties or to properties of an expression without properties (`null` or `undefined`).

Destructuring

For more complex assignments, the destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

```
const foo = ['one', 'two', 'three'];

// without destructuring
const one   = foo[0];
```

```
const two   = foo[1];
const three = foo[2];

// with destructuring
const [one, two, three] = foo;
```

Evaluation and nesting

In general, assignments are used within a variable declaration (i.e., with `const`, `let`, or `var`) or as standalone statements).

```
// Declares a variable x and initializes it to
the result of f().
// The result of the x = f() assignment
expression is discarded.
let x = f();

x = g(); // Reassigns the variable x to the
result of g().
```

However, like other expressions, assignment expressions like `x = f()` evaluate into a result value. Although this result value is usually not used, it can then be used by another expression.

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, some JavaScript style guides discourage chaining or nesting assignments). Nevertheless, assignment chaining and nesting may occur sometimes, so it is important to be able to understand how they work.

By chaining or nesting an assignment expression, its result can itself be assigned to

another variable. It can be logged, it can be put inside an array literal or function call, and so on.

```
let x;
const y = (x = f()); // Or equivalently: const y = x = f();
console.log(y); // Logs the return value of the assignment x = f().

console.log(x = f()); // Logs the return value directly.

// An assignment expression can be nested in any place
// where expressions are generally allowed,
// such as array literals' elements or as function calls' arguments.
console.log([ 0, x = f(), 0 ]);
console.log(f(0, x = f(), 0));
```

The evaluation result matches the expression to the right of the `=` sign in the "Meaning" column of the table above. That means that `x = f()` evaluates into whatever `f()`'s result is, `x += f()` evaluates into the resulting sum `x + f()`, `x **= f()` evaluates into the resulting power `x ** y`, and so on.

In the case of logical assignments, `x &&= f()`, `x ||= f()`, and `x ??= f()`, the return value is that of the logical operation without the assignment, so `x && f()`, `x || f()`, and `x ?? f()`, respectively.

When chaining these expressions without parentheses or other grouping operators like array literals, the assignment expressions are **grouped right to left** (they are right-

associative), but they are **evaluated left to right**.

Note that, for all assignment operators other than `=` itself, the resulting values are always based on the operands' values *before* the operation.

For example, assume that the following functions `f` and `g` and the variables `x` and `y` have been declared:

```
function f () {
  console.log('F!');
  return 2;
}
function g () {
  console.log('G!');
  return 3;
}
let x, y;
```

Consider these three examples:

```
y = x = f()
y = [ f(), x = g() ]
x[f()] = g()
```

Evaluation example 1

`y = x = f()` is equivalent to `y = (x = f())`, because the assignment operator `=` is right-associative . However, it evaluates from left to right:

1. The assignment expression `y = x = f()` starts to evaluate.

i. The `y` on this assignment's left-hand side evaluates into a reference to the variable named `y`.

ii. The assignment expression `x = f()` starts to evaluate.

i. The `x` on this assignment's left-hand side evaluates into a reference to the variable named `x`.

ii. The function call `f()` prints "F!" to the console and then evaluates to the number `2`.

iii. That `2` result from `f()` is assigned to `x`.

iii. The assignment expression `x = f()` has now finished evaluating; its result is the new value of `x`, which is `2`.

iv. That `2` result in turn is also assigned to `y`.

2. The assignment expression `y = x = f()` has now finished evaluating; its result is the new value of `y` – which happens to be `2`. `x` and `y` are assigned to `2`, and the console has printed "F!".

Evaluation example 2

`y = [ f(), x = g() ]` also evaluates from left to right:

1. The assignment expression `y = [ f(), x = g() ]` starts to evaluate.

i. The `y` on this assignment's left-hand evaluates into a reference to the variable named `y`.

ii. The inner array literal `[ f(), x = g() ]` starts to evaluate.

i. The function call `f()` prints "F!" to the console and then evaluates to the number `2`.

ii. The assignment expression `x = g()` starts to evaluate.

i. The `x` on this assignment's left-hand side evaluates into a reference to the variable named `x`.

ii. The function call `g()` prints "G!" to the console and then evaluates to the number `3`.

iii. That `3` result from `g()` is assigned to `x`.

iii. The assignment expression `x = g()` has now finished evaluating; its result is the new value of `x`, which is `3`. That `3` result becomes the next element in the inner array literal (after the `2` from the `f()`).

iii. The inner array literal `[ f(), x = g() ]` has now finished evaluating; its result is an array with two values: `[ 2, 3 ]`.

iv. That `[ 2, 3 ]` array is now assigned to `y`.

2. The assignment expression `y = [ f(), x = g() ]` has now finished evaluating; its result is the new value of `y` – which happens to be `[ 2, 3 ]`. `x` is now assigned to `3`, `y` is now assigned to `[ 2,`

`3 ]` , and the console has printed "F!" then "G!".

Evaluation example 3

`x[f()] = g()` also evaluates from left to right. (This example assumes that `x` is already assigned to some object. For more information about objects, read Working with Objects.)

1. The assignment expression `x[f()] = g()` starts to evaluate.
    i. The `x[f()]` property access on this assignment's left-hand starts to evaluate.
        i. The `x` in this property access evaluates into a reference to the variable named `x` .
        ii. Then the function call `f()` prints "F!" to the console and then evaluates to the number `2` .
    ii. The `x[f()]` property access on this assignment has now finished evaluating; its result is a variable property reference: `x[2]` .
    iii. Then the function call `g()` prints "G!" to the console and then evaluates to the number `3` .
    iv. That `3` is now assigned to `x[2]` . (This step will succeed only if `x` is assigned to an object.)
2. The assignment expression `x[f()] = g()` has now finished evaluating; its result is the new value of `x[2]` – which happens to

be `3`. `x[2]` is now assigned to `3`, and the console has printed "F!" then "G!".

Avoid assignment chains

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, chaining assignments in the same statement is discouraged).

In particular, putting a variable chain in a `const`, `let`, or `var` statement often does *not* work. Only the outermost/leftmost variable would get declared; other variables within the assignment chain are *not* declared by the `const` / `let` / `var` statement. For example:

```
const z = y = x = f();
```

This statement seemingly declares the variables `x`, `y`, and `z`. However, it only actually declares the variable `z`. `y` and `x` are either invalid references to nonexistent variables (in strict mode) or, worse, would implicitly create global variables for `x` and `y` in sloppy mode.

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of

the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behavior generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the `===` and `!==` operators, which perform strict equality and inequality comparisons. These operators do not attempt to convert the operands to compatible types before checking equality. The following table describes the comparison operators in terms of this sample code:

```
const var1 = 3;
const var2 = 4;
```

Comparison operators

Operator	Description	Examples returning true

Equal (`==`)	Returns `true` if the operands are equal.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Not equal (`!=`)	Returns `true` if the operands are not equal.	`var1 != 4` `var2 != "3"`
Strict equal (`===`)	Returns `true` if the operands are equal and of the same type. See also	`3 === var1`

Operator	Description	Examples returning true
	`Object.is` and sameness in JS.	
Strict not equal (`!==`)	Returns `true` if the operands are of the same type but not equal, or are of different type.	`var1 !== "3"` `3 !== '3'`
Greater than (`>`)	Returns `true` if the left operand is greater than the right operand.	`var2 > var1` `"12" > 2`
Greater than or equal (`>=`)	Returns `true` if the left operand is greater than or equal to the right operand.	`var2 >= var1` `var1 >= 3`
Less than (`<`)	Returns `true` if the left operand is less than the right operand.	`var1 < var2` `"2" < 12`
Less than or equal (`<=`)	Returns `true` if the left operand is less than or equal to the right operand.	`var1 <= var2` `var2 <= 5`

> **Note:** `=>` is not a comparison operator but rather is the notation for

> Arrow functions.

Arithmetic operators

An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces `Infinity`). For example:

```
1 / 2; // 0.5
1 / 2 === 1.0 / 2.0; // this is true
```

In addition to the standard arithmetic operations (`+` , `-` , `*` , `/`), JavaScript provides the arithmetic operators listed in the following table:

Arithmetic operators

Operator	Description	Example
Remainder (`%`)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.

Operator	Description	Example
Increment (++)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then,

Operator	Description	Example
		sets `x` to 2.
Unary negation (`-`)	Unary operator. Returns the negation of its operand.	If `x` is 3, then `-x` returns -3.
Unary plus (`+`)	Unary operator. Attempts to convert the operand to a number, if it is not already.	`+"3"` returns `3`. `+true` returns `1`.
Exponentiation operator (`**`)	Calculates the `base` to the `exponent` power, that is, `base^exponent`	`2 ** 3` returns `8`. `10 ** -1` returns `0.1`.

Bitwise operators

A bitwise operator treats their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators.

Operator	Usage	Description
Bitwise AND	`a & b`	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	`a | b`	Returns a zero in each bit position for which the corresponding bits of both operands are zeros.
Bitwise XOR	`a ^ b`	Returns a zero in each bit position for which the corresponding bits are the same. [Returns a one in each bit position for which the corresponding bits are different.]
Bitwise NOT	`~ a`	Inverts the bits of its operand.
Left shift	`a << b`	Shifts `a` in binary representation `b` bits to the left,

Operator	Usage	Description
		shifting in zeros from the right.
Sign-propagating right shift	`a >> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off.
Zero-fill right shift	`a >>> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones). Numbers with more than 32 bits get their most significant bits discarded. For example, the following integer with more than 32 bits will be converted to a 32-bit integer:

```
Before: 1110 0110 1111 1010 0000 0000 0000
0110 0000 0000 0001
After:                 1010 0000 0000 0000
0110 0000 0000 0001
```

- Each bit in the first operand is paired with the corresponding bit in the second

operand: first bit to first bit, second bit to second bit, and so on.

- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

Expression	Result	Binary Description
`15 & 9`	`9`	`1111 & 1001 = 1001`
`15 | 9`	`15`	`1111 | 1001 = 1111`
`15 ^ 9`	`6`	`1111 ^ 1001 = 0110`
`~15`	`-16`	`~ 0000 0000 … 0000 1111 = 1111 1111 … 1111 0000`
`~9`	`-10`	`~ 0000 0000 … 0000 1001 = 1111 1111 … 1111 0110`

Note that all 32 bits are inverted using the Bitwise NOT operator, and that values with the most significant (left-most) bit set to 1 represent negative numbers (two's-complement representation). `~x` evaluates to the same value that `-x - 1` evaluates to.

Bitwise shift operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the

second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of either type `Number` or `BigInt`: specifically, if the type of the left operand is `BigInt`, they return `BigInt`; otherwise, they return `Number`.

The shift operators are listed in the following table.

Bitwise shift operators

Operator	Description	Example
Left shift (<<)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	`9<<2` yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.

Operator	Description	Example
Sign-propagating right shift (>>)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	`9>>2` yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, `-9>>2` yields -3, because the sign is preserved.
Zero-fill right shift (>>>)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded.	`19>>>2` yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill

Operator	Description	Example
	Zero bits are shifted in from the left.	right shift and sign-propagating right shift yield the same result.

Logical operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the `&&` and `||` operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Logical operators

Operator	Usage	Description
Logical AND (`&&`)	`expr1 && expr2`	Returns `expr1` if it can be converted to `false` ; otherwise, returns `expr2` . Thus, when used with Boolean values, `&&` returns `true` if both

Logical OR (`||`)	`expr1 || `	Returns `expr1` if it can be converted to

Operator	Usage	Description
	`expr2`	`true`; otherwise, returns `expr2`. Thus, when used with Boolean values, `||` returns `true` if either operand is true; if both are false, returns `false`.
Logical NOT (`!`)	`!expr`	Returns `false` if its single operand that can be converted to `true`; otherwise, returns `true`.

Examples of expressions that can be converted to `false` are those that evaluate to null, 0, NaN, the empty string (""), or undefined.

The following code shows examples of the `&&` (logical AND) operator.

```
const a1 =  true && true; // t && t returns
true
const a2 =  true && false; // t && f returns
false
const a3 = false && true; // f && t returns
false
const a4 = false && (3 === 4); // f && f
returns false
const a5 = 'Cat' && 'Dog'; // t && t returns
Dog
const a6 = false && 'Cat'; // f && t returns
false
const a7 = 'Cat' && false; // t && f returns
false
```

The following code shows examples of the || (logical OR) operator.

```
const o1 =  true || true; // t || t returns true
const o2 = false || true; // f || t returns true
const o3 =  true || false; // t || f returns true
const o4 = false || (3 === 4); // f || f returns false
const o5 = 'Cat' || 'Dog'; // t || t returns Cat
const o6 = false || 'Cat'; // f || t returns Cat
const o7 = 'Cat' || false; // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
const n1 = !true; // !t returns false
const n2 = !false; // !f returns true
const n3 = !'Cat'; // !t returns false
```

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false && anything` is short-circuit evaluated to false.
- `true || anything` is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not

evaluated, so any side effects of doing so do not take effect.

Note that for the second case, in modern code you can use the Nullish coalescing operator (`??`) that works like `||` , but it only returns the second expression, when the first one is "nullish", i.e. `null` or `undefined` . It is thus the better alternative to provide defaults, when values like `''` or `0` are valid values for the first expression, too.

BigInt operators

Most operators that can be used between numbers can be used between `BigInt` values as well.

```
// BigInt addition
const a = 1n + 2n; // 3n
// Division with BigInts round towards zero
const b = 1n / 2n; // 0n
// Bitwise operations with BigInts do not
truncate either side
const c = 40000000000000000n >> 2n; //
10000000000000000n
```

One exception is unsigned right shift (`>>>`), which is not defined for BigInt values. This is because a BigInt does not have a fixed width, so technically it does not have a "highest bit".

```
const d = 8n >>> 2n; // TypeError: BigInts have
no unsigned right shift, use >> instead
```

BigInts and numbers are not mutually replaceable — you cannot mix them in calculations.

```
const a = 1n + 2; // TypeError: Cannot mix
BigInt and other types
```

This is because BigInt is neither a subset nor a superset of numbers. BigInts have higher precision than numbers when representing large integers, but cannot represent decimals, so implicit conversion on either side might lose precision. Use explicit conversion to signal whether you wish the operation to be a number operation or a BigInt one.

```
const a = Number(1n) + 2; // 3
const b = 1n + BigInt(2); // 3n
```

You can compare BigInts with numbers.

```
const a = 1n > 2; // false
const b = 3 > 2n; // true
```

String operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

For example,

```
console.log('my ' + 'string'); // console logs
the string "my string".
```

The shorthand assignment operator `+=` can also be used to concatenate strings.

For example,

```
let mystring = 'alpha';
mystring += 'bet'; // evaluates to "alphabet"
and assigns this value to mystring.
```

Conditional (ternary) operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

```
condition ? val1 : val2
```

If `condition` is true, the operator has the value of `val1`. Otherwise it has the value of `val2`. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
const status = age >= 18 ? 'adult' : 'minor';
```

This statement assigns the value "adult" to the variable `status` if `age` is eighteen or more. Otherwise, it assigns the value "minor" to `status`.

Comma operator

The comma operator (,) evaluates both of its operands and returns the value of the last

operand. This operator is primarily used inside a `for` loop, to allow multiple variables to be updated each time through the loop. It is regarded bad style to use it elsewhere, when it is not necessary. Often two separate statements can and should be used instead.

For example, if `a` is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to update two variables at once. The code prints the values of the diagonal elements in the array:

```
const x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const a = [x, x, x, x, x];

for (let i = 0, j = 9; i <= j; i++, j--) {
//                                 ^
  console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

Unary operators

A unary operation is an operation with only one operand.

delete

The `delete` operator deletes an object's property. The syntax is:

```
delete object.property;
delete object[propertyKey];
delete objectName[index];
```

where `object` is the name of an object, `property` is an existing property, and

`propertyKey` is a string or symbol referring to an existing property.

If the `delete` operator succeeds, it removes the property from the object. Trying to access it afterwards will yield `undefined`. The `delete` operator returns `true` if the operation is possible; it returns `false` if the operation is not possible.

```
delete Math.PI; // returns false (cannot delete non-configurable properties)

const myObj = {h: 4};
delete myObj.h; // returns true (can delete user-defined properties)
```

Deleting array elements

Since arrays are just objects, it's technically possible to `delete` elements from them. This is however regarded as a bad practice, try to avoid it. When you delete an array property, the array length is not affected and other elements are not re-indexed. To achieve that behavior, it is much better to just overwrite the element with the value `undefined`. To actually manipulate the array, use the various array methods such as `splice`.

typeof

The `typeof` operator is used in either of the following ways:

```
typeof operand
```

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

```
const myFun = new Function('5 + 2');
const shape = 'round';
const size = 1;
const foo = ['Apple', 'Mango', 'Orange'];
const today = new Date();
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun;       // returns "function"
typeof shape;       // returns "string"
typeof size;        // returns "number"
typeof foo;         // returns "object"
typeof today;       // returns "object"
typeof doesntExist; // returns "undefined"
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true; // returns "boolean"
typeof null; // returns "object"
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62;            // returns "number"
typeof 'Hello world'; // returns "string"
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified; // returns
"string"
typeof window.length;         // returns
"number"
typeof Math.LN2;              // returns
"number"
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur;        // returns "function"
typeof eval;        // returns "function"
typeof parseInt;    // returns "function"
typeof shape.split; // returns "function"
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date;     // returns "function"
typeof Function; // returns "function"
typeof Math;     // returns "object"
typeof Option;   // returns "function"
typeof String;   // returns "function"
```

void

The `void` operator is used in either of the following ways:

```
void (expression)
void expression
```

The `void` operator specifies an expression to be evaluated without returning a value.

`expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

Relational operators

A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.

in

The `in` operator returns `true` if the specified property is in the specified object. The syntax is:

```
propNameOrNumber in objectName
```

where `propNameOrNumber` is a string, numeric, or symbol expression representing a property name or array index, and `objectName` is the name of an object.

The following examples show some uses of the `in` operator.

```
// Arrays
const trees = ['redwood', 'bay', 'cedar',
'oak', 'maple'];
0 in trees;        // returns true
3 in trees;        // returns true
6 in trees;        // returns false
'bay' in trees;    // returns false (you must
specify the index number,
                   // not the value at that
index)
'length' in trees; // returns true (length is
an Array property)
```

```
// built-in objects
'PI' in Math;          // returns true
const myString = new String('coral');
'length' in myString;  // returns true

// Custom objects
const mycar = { make: 'Honda', model: 'Accord',
year: 1998 };
'make' in mycar;  // returns true
'model' in mycar; // returns true
```

instanceof

The `instanceof` operator returns `true` if the specified object is of the specified object type. The syntax is:

```
objectName instanceof objectType
```

where `objectName` is the name of the object to compare to `objectType`, and `objectType` is an object type, such as `Date` or `Array`.

Use `instanceof` when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

For example, the following code uses `instanceof` to determine whether `theDay` is a `Date` object. Because `theDay` is a `Date` object, the statements in the `if` statement execute.

```
const theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
  // statements to execute
}
```

Basic expressions

All operators eventually operate on one or more basic expressions. These basic expressions include identifiers and literals, but there are a few other kinds as well. They are briefly introduced below, and their semantics are described in detail in their respective reference sections.

this

Use the `this` keyword to refer to the current object. In general, `this` refers to the calling object in a method. Use `this` either with the dot or the bracket notation:

```
this['propertyName']
this.propertyName
```

Suppose a function called `validate` validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
  if ((obj.value < lowval) || (obj.value >
hival)) {
    console.log('Invalid Value!');
  }
}
```

You could call `validate` in each form element's `onChange` event handler, using `this` to pass it to the form element, as in the following example:

```
<p>Enter a number between 18 and 99:</p>
<input type="text" name="age" size="3"
onChange="validate(this, 18, 99);" />
```

Grouping operator

The grouping operator `( )` controls the precedence of evaluation in expressions. For example, you can override multiplication and division first, then addition and subtraction to evaluate addition first.

```
const a = 1;
const b = 2;
const c = 3;

// default precedence
a + b * c     // 7
// evaluated by default like this
a + (b * c)   // 7

// now overriding precedence
// addition before multiplication
(a + b) * c   // 9

// which is equivalent to
a * c + b * c // 9
```

new

You can use the `new` operator to create an instance of a user-defined object type or of one of the built-in object types. Use `new` as follows:

```
const objectName = new objectType(param1,
param2, /* …, */ paramN);
```

super

The `super` keyword is used to call functions on an object's parent. It is useful with classes to call the parent constructor, for example.

```
super([arguments]); // calls the parent
constructor.
super.functionOnParent([arguments]);
```

Last modified: Nov 23, 2022, by MDN contributors