# Expressions and operators

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`.

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 ().

JavaScript has both binary and unary operators, and one special ternary operator, the conditional operator.
A **unary** operator requires a single operand, either before or after the operator. Also `delete, typeof, void` are type of unary operators.

```
i++;
++i;
```

A **binary** operator requires two operands, one before the operator and one after the operator:

```
a * b;
2 + 4;
```

While a **ternary** operator is a **conditional** operator that requires three operands.

```
var numtype = (num%2==0) ? "even": "odd";
```

## 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.

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 assignment | `x >>= f()` | `x = x >> 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 }
```

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.
```

## Destructuring

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.

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
```

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.

## Avoid assignment chains

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.

The following table describes the comparison operators in terms of this sample code:

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

| Operator | Description | Examples returning true |
|---|---|---|
| Equal (==) | Returns `true` if the operands are equal. | `3 == var1`<br>`"3" == var1`<br>`3 == '3'` |
| Not equal (!=) | Returns `true` if the operands are not equal. | `var1 != 4`<br>`var2 != "3"` |
| Strict equal (===) | Returns `true` if the operands are equal and of the same type. | `3 === var1` |
| Strict not equal (!==) | Returns `true` if the operands are of the same type but not equal, or are of a different type. | `var1 !== "3"`<br>`3 !== '3'` |
| Greater than (>) | Returns `true` if the left operand is greater than the right operand. | `var2 > var1`<br>`"12" > 2` |
| Greater than or equal (>=) | Returns `true` if the left operand is greater than or equal to the right operand. | `var2 >= var1`<br>`var1 >= 3` |
| Less than (<) | Returns `true` if the left operand is less than the right operand. | `var1 < var2`<br>`"2" < 12` |
| Less than or equal (<=) | Returns `true` if the left operand is less than or equal to the right operand. | `var1 <= var2`<br>`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 (/).

```
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:

| **Operator** | **Description** | **Example** |
|---|---|---|
| Remainder (%) | Binary operator. Returns the integer remainder of dividing the two operands. | 12 % 5 returns 2. |
| 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, 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`.<br>`+true` returns `1`. |
| Exponentiation operator (**) | Calculates the base to the exponent power, that is, base^exponent | `2 ** 3` returns `8`.<br>`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. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

JavaScript has six bitwise operators that allow you to manipulate the binary representation of numbers at the bit level. Here are the bitwise operators in JavaScript:

1. Bitwise AND (&) - Returns a 1 in each bit position for which the corresponding bits of both operands are 1.
2. Bitwise OR (|) - Returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.
3. Bitwise XOR (**^**) - Returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1.
4. Bitwise NOT (**~**) - Inverts the bits of its operand. In other words, it changes every 0 to a 1 and every 1 to a 0.
5. Left shift (**<<**) - Shifts the bits of the left-hand operand to the left by the number of positions specified by the right-hand operand.
6. Right shift (**>>**) - Shifts the bits of the left-hand operand to the right by the number of positions specified by the right-hand operand.

Here's an example of using bitwise operators in JavaScript:

```
let a = 5; // binary representation: 0101
let b = 3; // binary representation: 0011

console.log(a & b); // 0001 (1 in binary)
console.log(a | b); // 0111 (7 in binary)
console.log(a ^ b); // 0110 (6 in binary)
console.log(~a); // -6 (in 2's complement representation)
console.log(a << 1); // 1010 (10 in binary)
console.log(a >> 1); // 0010 (2 in binary)
```

## 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.

| **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 operands are true; otherwise, returns `false`. |
| Logical OR (\|\|) | `expr1 \|\| expr2` | Returns `expr1` if it can be converted to `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`. |

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.

## 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".

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.

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];
```

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)
```

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

In JavaScript, the `typeof` operator is used to determine the type of a given value or expression. It returns a string that specifies the data type of the operand.

The syntax for `typeof` is as follows:

```
typeof operand
```

Here, the `operand` can be any valid JavaScript expression.

The possible return values for `typeof` are:

- "undefined" - if the operand is an undefined value
- "boolean" - if the operand is a boolean value
- "number" - if the operand is a number
- "string" - if the operand is a string
- "bigint" - if the operand is a BigInt
- "symbol" - if the operand is a symbol
- "function" - if the operand is a function
- "object" - if the operand is an object or null

It's worth noting that `typeof null` returns "`object`", which is a quirk of JavaScript that has been present since the early days of the language and cannot be changed for compatibility reasons.

### void

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 to avoid precedence issues.

## 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, 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

`this` is a keyword in JavaScript that refers to the current object that a method or function is a property of. The value of `this` depends on how the method or function is called.

Here are a few common ways that `this` is used in JavaScript:

1. Global context In the global context, `this` refers to the global object, which is `window` in a web browser or `global` in Node.js.
2. Function context When `this` is used inside a function, it refers to the object that the function is a method of. For example:

```
const obj = { name: "John", sayName: function() { console.log(this.name); } };
 obj.sayName(); // logs "John"
```

In this example, `this` inside the `sayName` function refers to the `obj` object.

3. Event handlers In event handlers, `this` refers to the DOM element that the event is attached to. For example:

```
const button = document.querySelector('button'); button.addEventListener('click',
function() { console.log(this); // logs the button element });
```

4. Constructor functions In constructor functions, `this` refers to the new object that is created when the constructor is called. For example:

```
function Person(name, age) { this.name = name; this.age = age; } const john = new
Person("John", 30); console.log(john.name); // logs "John"
```

In this example, `this` inside the `Person` function refers to the new object that is created when `new Person()` is called.

These are just a few examples of how `this` is used in `JavaScript`. The behavior of `this` can be complex and can depend on a number of factors, so it's important to understand its context and usage in different scenarios.

## 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(args); // calls the parent constructor.
super.functionOnParent(args);
```