**Operators**

Many operators are known to us from school. It is an addition +, a multiplication \*, a substraction - and so on.

[**Terms: “unary”, “binary”, “operand”**](https://javascript.info/operators#terms-unary-binary-operand)

Before we move on, let’s grasp the common terminology.

* *An operand* – is what operators are applied to. For instance in multiplication 5 \* 2 there are two operands: the left operand is 5, and the right operand is 2. Sometimes people say “arguments” instead of “operands”.
* An operator is *unary* if it has a single operand. For example, the unary minus "-" reverses the sign of the number:

let x = 1;

x = -x;

alert( x ); // -1, unary minus was applied

* An operator is *binary* if it has two operands. The same minus exists in the binary form as well:
* let x = 1, y = 3;

alert( y - x ); // 2, binary minus substracts values

Formally, we’re talking about the two different operators here: the unary minus (single operand, reverses the sign) and the binary minus (two operands, substracts).

**[Strings concatenation, binary +](https://javascript.info/operators" \l "strings-concatenation-binary)**

Now let’s see special features of JavaScript operators that are beyond school arithmetics.

Usually the plus operator '+' sums numbers.

But if the binary + is applied to strings, it merges (concatenates) them:

let s = "my" + "string";

alert(s); // mystring

Note that if any of operands is a string, then the other one is converted to string too.

For example:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

See, it doesn’t matter whether the first operand is a string or the second one. The rule is simple: if any of operands is a string, then convert the other one into a string as well.

The string concatenation and conversion is the special feature of the binary plus "+". Other arithmetic operators work only with numbers. They always convert their operands to numbers.

For instance, subtraction and division:

alert( 2 - '1' ); // 1

alert( '6' / '2' ); // 3

**[Numeric conversion, unary +](https://javascript.info/operators" \l "numeric-conversion-unary)**

The plus + exist in two forms. The binary form that we used above and the unary form.

The unary plus or, in other words, the plus operator + applied to a single value, doesn’t do anything with numbers, but if the operand is not a number, then it is converted into it.

For example:

// No effect on numbers

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Converts non-numbers

alert( +true ); // 1

alert( +"" ); // 0

It actually does the same as Number(...), but shorter.

A need to convert string to number arises very often. For example, if we are getting values from HTML form fields, they are usually strings.

What if we want to sum them?

The binary plus would add them as strings:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23", the binary plus concatenates strings

If we want to treat them as numbers, then we can convert and then sum:

let apples = "2";

let oranges = "3";

// both values converted to numbers before the binary plus

alert( +apples + +oranges ); // 5

// the longer variant

// alert( Number(apples) + Number(oranges) ); // 5

From a mathematician’s standpoint the abundance of pluses may seem strange. But from a programmer’s standpoint – there’s nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up.

Why are unary pluses applied to values before the binary one? As we’re going to see, that’s because of their *higher precedence*.

**[Operators precedence](https://javascript.info/operators" \l "operators-precedence)**

If an expression has more than one operator, the execution order is defined by their *precedence*, or, in other words, there’s an implicit priority order among the operators.

From the school we all know that the multiplication in the expression 1 + 2 \* 2 should be calculated before the addition. That’s exactly the precedence thing. The multiplication is said to have *a higher precedence* than the addition.

Parentheses override any precedence, so if we’re not satisfied with the order, we can use them, like: (1 + 2) \* 2.

There are many operators in JavaScript. Every operator has a corresponding precedence number. The one with the bigger number executes first. If the precedence is same – the execution order is from left to right.

An extract from the [precedence table](https://developer.mozilla.org/en/JavaScript/Reference/operators/operator_precedence):

| **Precedence** | **Name** | **Sign** |
| --- | --- | --- |
| … | … | … |
| 15 | unary plus | + |
| 15 | unary minus | - |
| 14 | multiplication | \* |
| 14 | division | / |
| 13 | addition (binary) | + |
| 13 | subtraction | - |
| … | … | … |
| 3 | assignment | = |
| … | … | … |

As we can see, the “unary plus” has a priority of 15, higher than 13 for the “addition” (binary plus). That’s why in the expression "+apples + +oranges" unary pluses work first, and then the addition.

**[Assignment](https://javascript.info/operators" \l "assignment)**

Let’s note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 3.

That’s why when we assign a variable, like x = 2 \* 2 + 1, then the calculations are done first, and afterwards the = is evaluated, storing the result in x.

let x = 2 \* 2 + 1;

alert( x ); // 5

**The assignment operator "=" returns a value**

An operator always returns a value. That’s obvious for most of them like an addition + or a multiplication \*. But the assignment operator follows that rule too.

The call x = value writes the value into x *and then returns it*.

Here’s the demo that uses an assignment as the part of a more complex expression:

let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0

In the example above, the result of (a = b + 1) is the value which is assigned to a (that is 3). It is then used to substract from 3.

Funny code, isn’t it? We should understand how it works, because sometimes we can see it in 3rd-party libraries, but shouldn’t write anything like that ourselves. Such tricks definitely don’t make the code clearer and readable.

**[Remainder %](https://javascript.info/operators" \l "remainder)**

The remainder operator % despite it’s look does not have a relation to percents.

The result of a % b is the remainder of the integer division of a by b.

For instance:

alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2

alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3

alert( 6 % 3 ); // 0 is a remainder of 6 divided by 3

**[Exponentiation \*\*](https://javascript.info/operators" \l "exponentiation)**

The exponentiation operator \*\* is a recent addition to the language.

For a natural number b, the result of a \*\* b is a multiplied by itself b times.

For instance:

alert( 2 \*\* 2 ); // 4 (2 \* 2)

alert( 2 \*\* 3 ); // 8 (2 \* 2 \* 2)

alert( 2 \*\* 4 ); // 16 (2 \* 2 \* 2 \* 2)

The operator works for non-integer numbers of a and b as well, for instance:

alert( 4 \*\* (1/2) ); // 2 (power of 1/2 is the same as a square root, that's maths)

alert( 8 \*\* (1/3) ); // 2 (power of 1/3 is the same as a cubic root)

**[Increment/decrement](https://javascript.info/operators" \l "increment-decrement)**

Increasing or decreasing a number by one is among the most common numerical operations.

So, there are special operators for that:

* **Increment** ++ increases a variable by 1:
* let counter = 2;
* counter++; // works same as counter = counter + 1, but shorter

alert( counter ); // 3

* **Decrement** -- decreases a variable by 1:
* let counter = 2;
* counter--; // works same as counter = counter - 1, but shorter

alert( counter ); // 1

**Important:**

Increment/decrement can be applied only to a variable. An attempt to use it on a value like 5++ will give an error.

Operators ++ and -- can be placed both after and before the variable.

* When the operator goes after the variable, it is called a “postfix form”: counter++.
* The “prefix form” is when the operator stands before the variable: ++counter.

Both of these records do the same: increase i by 1.

Is there any difference? Yes, but we can only see it if we use the returned value of ++/--.

Let’s clarify. As we know, all operators return a value. Increment/decrement is not an exception here. The prefix form returns the new value, while the postfix form returns the old value (prior to increment/decrement).

To see the difference – here’s the example:

let counter = 1;

let a = ++counter; // (\*)

alert(a); // 2

Here in the line (\*) the prefix call ++counter increments i and returns the new value that is 2. So the alert shows 2.

Now let’s use the postfix form:

let counter = 1;

let a = counter++; // (\*) changed ++counter to counter++

alert(a); // 1

In the line (\*) the *postfix* form counter++ also increments i, but returns the *old* value (prior to increment). So the alert shows 1.

To summarize:

* If the result of increment/decrement is not used, then there is no difference which form to use:
* let counter = 0;
* counter++;
* ++counter;

alert( counter ); // 2, the lines above did the same

* If we’d like to increase the value *and* use the result of the operator right now, then we need the prefix form:
* let counter = 0;

alert( ++counter ); // 1

* If we’d like to increment, but use the previous value, then we need the postfix form:
* let counter = 0;

alert( counter++ ); // 0

**Increment/decrement among other operators**

Operators ++/-- can be used inside an expression as well. Their precedence is higher than most other arithmetical operations.

For instance:

let counter = 1;

alert( 2 \* ++counter ); // 4

Compare with:

let counter = 1;

alert( 2 \* counter++ ); // 2, because counter++ returns the "old" value

Though technically allowable, such notation usually makes the code less readable. One line does multiple things – not good.

While reading the code, a fast “vertical” eye-scan can easily miss such counter++, and it won’t be obvious that the variable increases.

The “one line – one action” style is advised:

let counter = 1;

alert( 2 \* counter );

counter++;

**[Modify-in-place](https://javascript.info/operators" \l "modify-in-place)**

We often need to apply an operator to a variable and store the new result in it.

For example:

let n = 2;

n = n + 5;

n = n \* 2;

This notation can be shortened using operators += and \*=:

let n = 2;

n += 5; // now n=7 (same as n = n + 5)

n \*= 2; // now n=14 (same as n = n \* 2)

alert( n ); // 14

Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: /=, -= etc.

Such operators have the same precedence as a normal assignment, so they run after most other calculations:

let n = 2;

n \*= 3 + 5;

alert( n ); // 16 (right part evaluated first, same as n \*= 8)

**Questions and Exercise**

1. What are the final values of all variables a, b, c and d after the code below?

let a = 1, b = 1;

let c = ++a; // ?

let d = b++; // ?

a=1 b=1 c=2 d=1

1. What are the values of a and x after the code below?

let a = 2;

let x = 1 + (a \*= 2);

a=2 x=5

1. Create a program that displays the volume of a cylinder with a height of 10cm and a diameter of 15cm.
2. Create a program that displays the surface area of a cube with a side length of 2.25m.
3. Create a program that determines how much money you would have in an investment if you invested $10,986.98 for 5 years at 4.3%. The formula for this is A=P(1+i)n, where P in your principal investment, i is the interest rate as a decimal and n is the number of years.