11th February 2020

Operators

We know many operators from school. They are things like addition `+`, multiplication `*`, subtraction `-`, and so on.

In this chapter, we'll concentrate on aspects of operators that are not covered by school arithmetic.

Terms: "unary", "binary", "operand"

Before we move on, let's grasp some common terminology.

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

```
let x = 1;

x = -x;
alert( x ); // -1, unary negation was applied
```

- An operator is *binary* if it has two operands. The same minus exists in binary form as well:

```
let x = 1, y = 3;
alert( y - x ); // 2, binary minus subtracts values
```

Formally, in the examples above we have two different operators that share the same symbol: the negation operator, a unary operator that reverses the sign, and the subtraction operator, a binary operator that subtracts one number from another.

String 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 one of the operands is a string, the other one is converted to a 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 either operand is a string, the other one is converted into a string as well.

However, note that operations run from left to right. If there are two numbers followed by a string, the numbers will be added before being converted to a string:

```
alert(2 + 2 + '1' ); // "41" and not "221"
```

String concatenation and conversion is a special feature of the binary plus `+`. Other arithmetic operators work only with numbers and always convert their operands to numbers.

For instance, subtraction and division:

```
alert( 2 - '1' ); // 1
alert( '6' / '2' ); // 3
```

Numeric conversion, unary +

The plus `+` exists 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 to numbers. But if the operand is not a number, the unary plus converts it into a number.

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 thing as `Number(...)`, but is shorter.

The need to convert strings to numbers 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, we need to convert and then sum them:

```
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 ones? As we're going to see, that's because of their *higher precedence*.

Operator precedence

If an expression has more than one operator, the execution order is defined by their *precedence*, or, in other words, the default priority order of operators.

From 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 default order, we can use them to change it. For example, write `(1 + 2) * 2`.

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

Here's an extract from the precedence table (you don't need to remember this, but note that unary operators are higher than corresponding binary ones):

Precedence	Name	Sign
…	…	…
17	unary plus	+
17	unary negation	-
15	multiplication	*

Precedence	Name	Sign
15	division	/
13	addition	+
13	subtraction	-
…	…	…
3	assignment	=
…	…	…

As we can see, the “unary plus” has a priority of `16` which is higher than the `13` of “addition” (binary plus). That’s why, in the expression `"+apples + +oranges"`, unary pluses work before the addition.

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`, the calculations are done first and then the `=` is evaluated, storing the result in `x`.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

It is possible to chain assignments:

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Chained assignments evaluate from right to left. First, the rightmost expression `2 + 2` is evaluated and then assigned to the variables on the left: `c`, `b` and `a`. At the end, all the variables share a single value.

The assignment operator "=" returns a value

An operator always returns a value. That's obvious for most of them like addition `+` or multiplication `*`. But the assignment operator follows this rule too.

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

Here's a demo that uses an assignment as 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 expression `(a = b + 1)` is the value which was assigned to `a` (that is `3`). It is then used for further evaluations.

Funny code, isn't it? We should understand how it works, because sometimes we see it in JavaScript libraries, but shouldn't write anything like that ourselves. Such tricks definitely don't make code clearer or readable.

Remainder %

The remainder operator `%`, despite its appearance, is not related 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 **

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 as well.

For instance:

```
alert( 4 ** (1/2) ); // 2 (power of 1/2 is the same as a square root, that's
alert( 8 ** (1/3) ); // 2 (power of 1/3 is the same as a cubic root)
```

Increment/decrement

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

So, there are special operators for it:

- **Increment** `++` increases a variable by 1:

```
let counter = 2;
counter++;        // works the same as counter = counter + 1, but is shorte
alert( counter ); // 3
```

- **Decrement** `--` decreases a variable by 1:

```
let counter = 2;
counter--;        // works the same as counter = counter - 1, but is shorte
alert( counter ); // 1
```

Important:

Increment/decrement can only be applied to variables. Trying to use it on a value like `5++` will give an error.

The operators `++` and `--` can be placed either before or after a variable.

- When the operator goes after the variable, it is in “postfix form”: `counter++`.
- The “prefix form” is when the operator goes before the variable: `++counter`.

Both of these statements do the same thing: increase `counter` 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 no exception. 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 an example:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

In the line `(*)`, the *prefix* form `++counter` increments `counter` and returns the new value, `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 `counter` but returns the *old* value (prior to increment). So, the `alert` shows `1`.

To summarize:

- If the result of increment/decrement is not used, there is no difference in which form to use:

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, the lines above did the same
```

- If we'd like to increase a value *and* immediately use the result of the operator, we need the prefix form:

```
let counter = 0;
alert( ++counter ); // 1
```

- If we'd like to increment a value but use its previous value, we need the postfix form:

```
let counter = 0;
alert( counter++ ); // 0
```

Increment/decrement among other operators

The operators `++/--` can be used inside expressions 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 okay, such notation usually makes code less readable. One line does multiple things – not good.

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

We advise a style of “one line – one action”:

```
let counter = 1;
alert( 2 * counter );
counter++;
```

Bitwise operators

Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation.

These operators are not JavaScript-specific. They are supported in most programming languages.

The list of operators:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

These operators are used very rarely. To understand them, we need to delve into low-level number representation and it would not be optimal to do that right now, especially since we won’t need them any time soon. If you’re curious, you can read the Bitwise Operators article on MDN. It would be more practical to do that when a real need arises.

Modify-in-place

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

For example:

```
let n = 2;
n = n + 5;
n = n * 2;
```

This notation can be shortened using the 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)
```

Comma

The comma operator `,` is one of the rarest and most unusual operators. Sometimes, it’s used to write shorter code, so we need to know it in order to understand what’s going on.

The comma operator allows us to evaluate several expressions, dividing them with a comma `,`. Each of them is evaluated but only the result of the last one is returned.

For example:

```
let a = (1 + 2, 3 + 4);

alert( a ); // 7 (the result of 3 + 4)
```

Here, the first expression `1 + 2` is evaluated and its result is thrown away. Then, `3 + 4` is evaluated and returned as the result.

Comma has a very low precedence

Please note that the comma operator has very low precedence, lower than `=`, so parentheses are important in the example above.

Without them: `a = 1 + 2, 3 + 4` evaluates `+` first, summing the numbers into `a = 3, 7`, then the assignment operator `=` assigns `a = 3`, and the rest is ignored. It's like `(a = 1 + 2), 3 + 4`.

Why do we need an operator that throws away everything except the last expression?

Sometimes, people use it in more complex constructs to put several actions in one line.

For example:

```
// three operations in one line
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

Such tricks are used in many JavaScript frameworks. That's why we're mentioning them. But usually they don't improve code readability so we should think well before using them.

Tasks

The postfix and prefix forms

importance: 5

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++; // ?
```

solution

Assignment result

importance: 3

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

```
let a = 2;

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

solution

Type conversions

importance: 5

What are results of these expressions?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
7 / 0
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Think well, write down and then compare with the answer.

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)

privacy policy