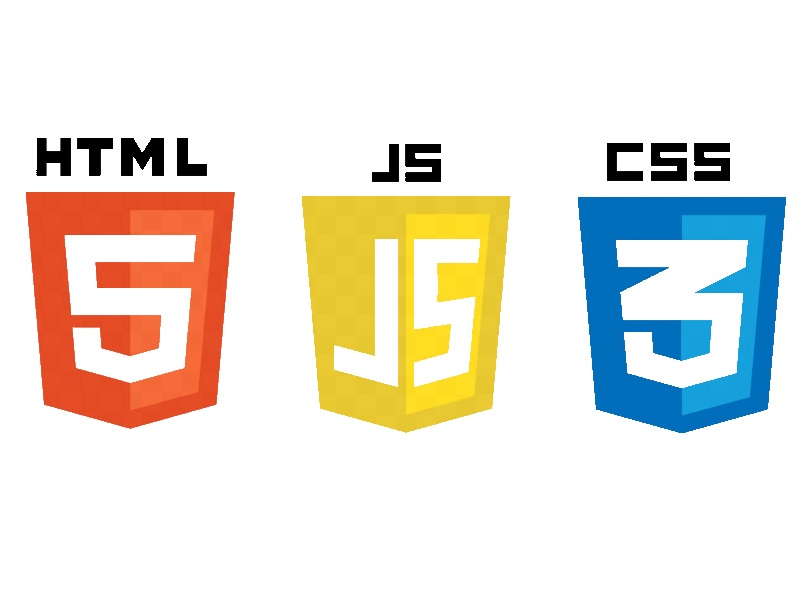
**A script for me**

**by me.**

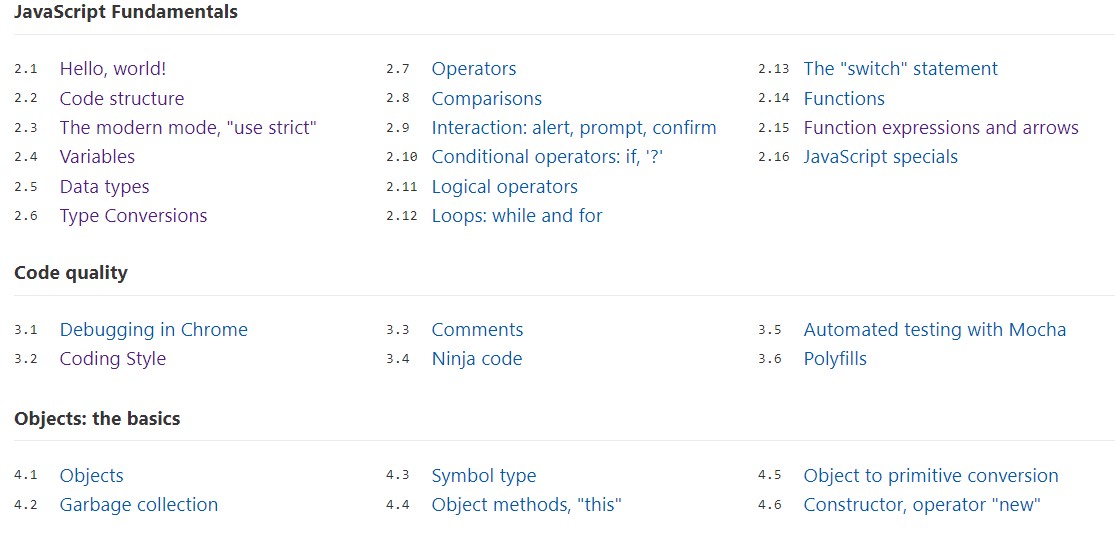
****

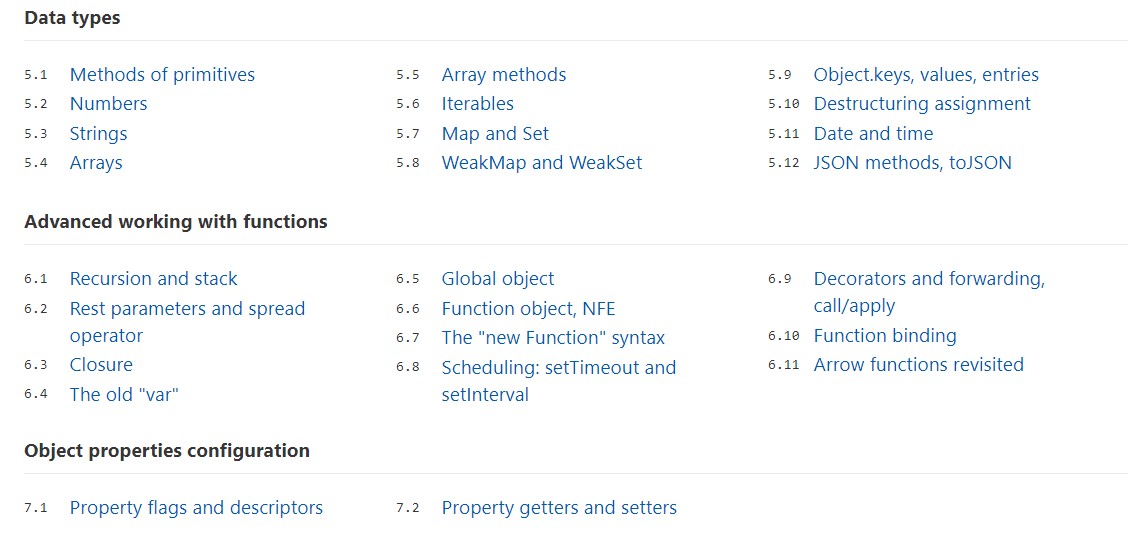
**DarjanDivkovic**

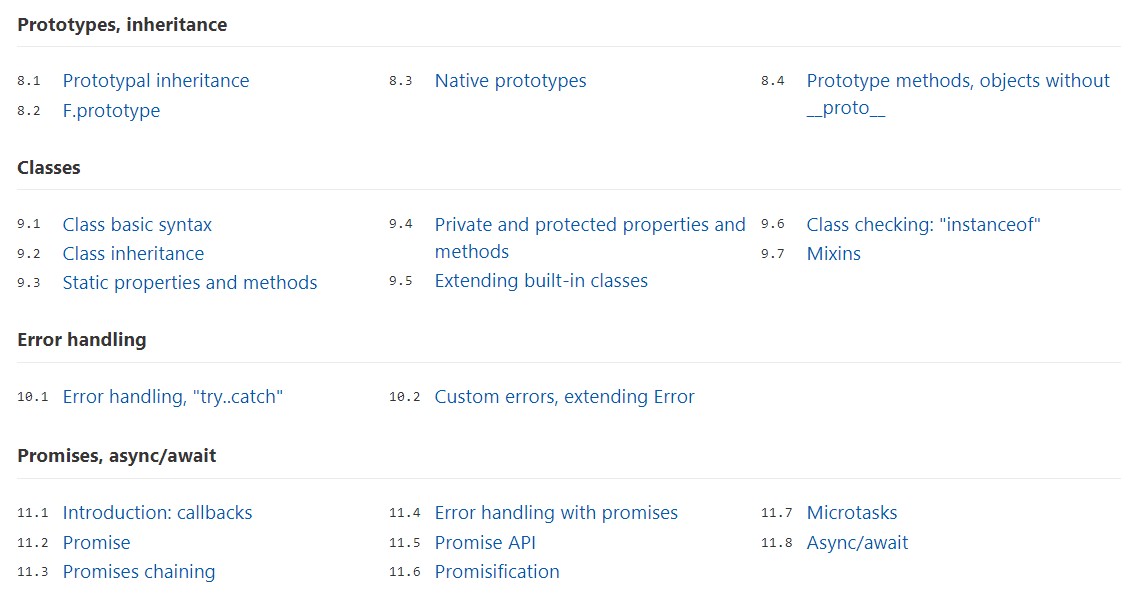
**The JavaScript language**

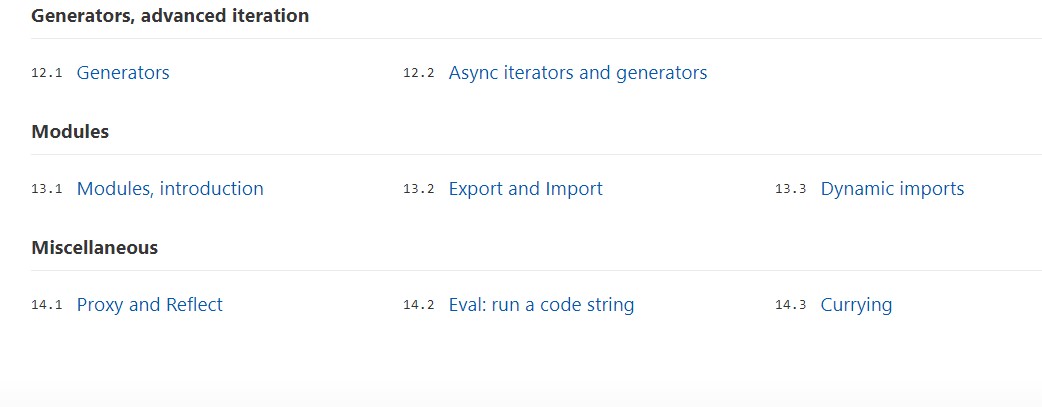
Here we learn JavaScript, starting from scratch and go on to advanced concepts like OOP.

We concentrate on the language itself here, with the minimum of environment-specific notes.









Hello World

* We can use a <script> tag to add JavaScript code to a page.
* The type and language attributes are not required.
* A script in an external file can be inserted with <script src="path/to/script.js"></script>.

There is much more to learn about browser scripts and their interaction with the webpage. But let’s keep in mind that this part of the tutorial is devoted to the JavaScript language, so we shouldn’t distract ourselves with browser-specific implementations of it. We’ll be using the browser as a way to run JavaScript, which is very convenient for online reading, but only one of many.

Always ''use strict'' [strict mode]

We have yet to cover the differences between strict mode and the “default” mode.

In the next chapters, as we learn language features, we’ll note the differences between the strict and default modes. Luckily, there aren’t many and they actually make our lives better.

For now, it’s enough to know about it in general:

1. The "use strict" directive switches the engine to the “modern” mode, changing the behavior of some built-in features. We’ll see the details later in the tutorial.
2. Strict mode is enabled by placing "use strict" at the top of a script or function. Several language features, like “classes” and “modules”, enable strict mode automatically.
3. Strict mode is supported by all modern browsers.
4. We recommended always starting scripts with "use strict". All examples in this tutorial assume strict mode unless (very rarely) specified otherwise.

Variables

We can declare variables to store data by using the var, let, or const keywords.

* let – is a modern variable declaration.
* var – is an old-school variable declaration. Normally we don’t use it at all, but we’ll cover subtle differences from let in the chapter [The old "var"](http://javascript.info/var), just in case you need them.
* const – is like let, but the value of the variable can’t be changed.

Variables should be named in a way that allows us to easily understand what’s inside them.

Data types

There are 7 basic data types in JavaScript.

* number for numbers of any kind: integer or floating-point.
* string for strings. A string may have one or more characters, there’s no separate single-character type.
* boolean for true/false.
* null for unknown values – a standalone type that has a single value null.
* undefined for unassigned values – a standalone type that has a single value undefined.
* object for more complex data structures.
* symbol for unique identifiers.

The typeof operator allows us to see which type is stored in a variable.

* Two forms: typeof x or typeof(x).
* Returns a string with the name of the type, like "string".
* For null returns "object" – this is an error in the language, it’s not actually an object.

In the next chapters, we’ll concentrate on primitive values and once we’re familiar with them, we’ll move on to objects.

Type Conversions

The three most widely used type conversions are to string, to number, and to boolean.

**String Conversion** – Occurs when we output something. Can be performed with String(value). The conversion to string is usually obvious for primitive values.

**Numeric Conversion** – Occurs in math operations. Can be performed with Number(value).

The conversion follows the rules:

| **Value** | **Becomes…** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | The string is read “as is”, whitespaces from both sides are ignored. An empty string becomes 0. An error gives NaN. |

**Boolean Conversion** – Occurs in logical operations. Can be performed with Boolean(value).

Follows the rules:

| **Value** | **Becomes…** |
| --- | --- |
| 0, null, undefined, NaN, "" | false |
| any other value | true |

Most of these rules are easy to understand and memorize. The notable exceptions where people usually make mistakes are:

* undefined is NaN as a number, not 0.
* "0" and space-only strings like " " are true as a boolean.

Objects aren’t covered here. We’ll return to them later in the chapter [Object to primitive conversion](http://javascript.info/object-toprimitive) that is devoted exclusively to objects after we learn more basic things about JavaScript.

Operators

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

## [Terms: “unary”, “binary”, “operand”](http://javascript.info/operators" \l "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

## [String concatenation, binary +](http://javascript.info/operators" \l "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

## [Assignment](http://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, 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.

## [Remainder %](http://javascript.info/operators" \l "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 \*\*](http://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 btimes.

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 maths)

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

## [Increment/decrement](http://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 it:

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

alert( counter ); // 3

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

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.
* 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](http://javascript.info/operators" \l "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](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) article on MDN. It would be more practical to do that when a real need arises.

## [Comma](http://javascript.info/operators" \l "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.

Comparison

* Comparison operators return a boolean value.
* Strings are compared letter-by-letter in the “dictionary” order.
* When values of different types are compared, they get converted to numbers (with the exclusion of a strict equality check).
* The values null and undefined equal == each other and do not equal any other value.
* Be careful when using comparisons like > or < with variables that can occasionally be null/undefined. Checking for null/undefinedseparately is a good idea.

Interaction: alert, prompt, confirm

We covered 3 browser-specific functions to interact with visitors:

**alert**

shows a message.

**prompt**

shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, null.

**confirm**

shows a message and waits for the user to press “OK” or “Cancel”. It returns true for OK and false for Cancel/Esc.

All these methods are modal: they pause script execution and don’t allow the visitor to interact with the rest of the page until the window has been dismissed.

There are two limitations shared by all the methods above:

1. The exact location of the modal window is determined by the browser. Usually, it’s in the center.
2. The exact look of the window also depends on the browser. We can’t modify it.

That is the price for simplicity. There are other ways to show nicer windows and richer interaction with the visitor, but if “bells and whistles” do not matter much, these methods work just fine.

## [Conditional operator ‘?’](http://javascript.info/ifelse" \l "conditional-operator)

Sometimes, we need to assign a variable depending on a condition.

For instance:

let accessAllowed;

let age = prompt('How old are you?', '');

if (age > 18) {

accessAllowed = true;

} else {

accessAllowed = false;

}

alert(accessAllowed);

The so-called “conditional” or “question mark” operator lets us do that in a shorter and simpler way.

The operator is represented by a question mark ?. Sometimes it’s called “ternary”, because the operator has three operands. It is actually the one and only operator in JavaScript which has that many.

The syntax is:

let result = condition ? value1 : value2;

The condition is evaluated: if it’s truthy then value1 is returned, otherwise – value2.

For example:

let accessAllowed = (age > 18) ? true : false;

Technically, we can omit the parentheses around age > 18. The question mark operator has a low precedence, so it executes after the comparison >.

This example will do the same thing as the previous one:

// the comparison operator "age > 18" executes first anyway

// (no need to wrap it into parentheses)

let accessAllowed = age > 18 ? true : false;

But parentheses make the code more readable, so we recommend using them.

**Please note:**

In the example above, you can avoid using the question mark operator because the comparison itself returns true/false:

// the same

let accessAllowed = age > 18;

## 

## [Multiple ‘?’](http://javascript.info/ifelse#multiple)

A sequence of question mark operators ? can return a value that depends on more than one condition.

For instance:

let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :

(age < 18) ? 'Hello!' :

(age < 100) ? 'Greetings!' :

'What an unusual age!';

alert( message );

It may be difficult at first to grasp what’s going on. But after a closer look, we can see that it’s just an ordinary sequence of tests:

1. The first question mark checks whether age < 3.
2. If true – it returns 'Hi, baby!'. Otherwise, it continues to the expression after the colon ‘":"’, checking age < 18.
3. If that’s true – it returns 'Hello!'. Otherwise, it continues to the expression after the next colon ‘":"’, checking age < 100.
4. If that’s true – it returns 'Greetings!'. Otherwise, it continues to the expression after the last colon ‘":"’, returning 'What an unusual age!'.

Here’s how this looks using if..else:

if (age < 3) {

message = 'Hi, baby!';

} else if (age < 18) {

message = 'Hello!';

} else if (age < 100) {

message = 'Greetings!';

} else {

message = 'What an unusual age!';

}

Logical operators

There are three logical operators in JavaScript: || (OR), && (AND), ! (NOT).

Although they are called “logical”, they can be applied to values of any type, not only boolean. Their result can also be of any type.

Let’s see the details.

## [|| (OR)](http://javascript.info/logical-operators" \l "or)

The “OR” operator is represented with two vertical line symbols:

result = a || b;

In classical programming, the logical OR is meant to manipulate boolean values only. If any of its arguments are true, it returns true, otherwise it returns false.

In JavaScript, the operator is a little bit trickier and more powerful. But first, let’s see what happens with boolean values.

There are four possible logical combinations:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

As we can see, the result is always true except for the case when both operands are false.

If an operand is not a boolean, it’s converted to a boolean for the evaluation.

For instance, the number 1 is treated as true, the number 0 as false:

if (1 || 0) { // works just like if( true || false )

alert( 'truthy!' );

}

Most of the time, OR || is used in an if statement to test if any of the given conditions is true.

For example:

let hour = 9;

if (hour < 10 || hour > 18) {

alert( 'The office is closed.' );

}

We can pass more conditions:

let hour = 12;

let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {

alert( 'The office is closed.' ); // it is the weekend

}

## [OR “||” finds the first truthy value](http://javascript.info/logical-operators" \l "or-finds-the-first-truthy-value)

The logic described above is somewhat classical. Now, let’s bring in the “extra” features of JavaScript.

The extended algorithm works as follows.

Given multiple OR’ed values:

result = value1 || value2 || value3;

The OR || operator does the following:

* Evaluates operands from left to right.
* For each operand, converts it to boolean. If the result is true, stops and returns the original value of that operand.
* If all operands have been evaluated (i.e. all were false), returns the last operand.

A value is returned in its original form, without the conversion.

In other words, a chain of OR "||" returns the first truthy value or the last one if no truthy value is found.

For instance:

alert( 1 || 0 ); // 1 (1 is truthy)

alert( true || 'no matter what' ); // (true is truthy)

alert( null || 1 ); // 1 (1 is the first truthy value)

alert( null || 0 || 1 ); // 1 (the first truthy value)

alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)

This leads to some interesting usage compared to a “pure, classical, boolean-only OR”.

1. **Getting the first truthy value from a list of variables or expressions.**

Imagine we have a list of variables which can either contain data or be null/undefined. How can we find the first one with data?

We can use OR ||:

let currentUser = null;

let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // selects "John" – the first truthy value

If both currentUser and defaultUser were falsy, "unnamed" would be the result.

1. **Short-circuit evaluation.**

Operands can be not only values, but arbitrary expressions. OR evaluates and tests them from left to right. The evaluation stops when a truthy value is reached, and the value is returned. This process is called “a short-circuit evaluation” because it goes as short as possible from left to right.

This is clearly seen when the expression given as the second argument has a side effect like a variable assignment.

In the example below, x does not get assigned:

let x;

true || (x = 1);

alert(x); // undefined, because (x = 1) not evaluated

If, instead, the first argument is false, || evaluates the second one, thus running the assignment:

let x;

false || (x = 1);

alert(x); // 1

An assignment is a simple case. There may be side effects, that won’t show up if the evaluation doesn’t reach them.

As we can see, such a use case is a "shorter way of doing if". The first operand is converted to boolean. If it’s false, the second one is evaluated.

Most of time, it’s better to use a “regular” if to keep the code easy to understand, but sometimes this can be handy.

## [&& (AND)](http://javascript.info/logical-operators" \l "and)

The AND operator is represented with two ampersands &&:

result = a && b;

In classical programming, AND returns true if both operands are truthy and false otherwise:

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

An example with if:

let hour = 12;

let minute = 30;

if (hour == 12 && minute == 30) {

alert( 'The time is 12:30' );

}

Just as with OR, any value is allowed as an operand of AND:

if (1 && 0) { // evaluated as true && false

alert( "won't work, because the result is falsy" );

}

## [AND “&&” finds the first falsy value](http://javascript.info/logical-operators" \l "and-finds-the-first-falsy-value)

Given multiple AND’ed values:

result = value1 && value2 && value3;

The AND && operator does the following:

* Evaluates operands from left to right.
* For each operand, converts it to a boolean. If the result is false, stops and returns the original value of that operand.
* If all operands have been evaluated (i.e. all were truthy), returns the last operand.

In other words, AND returns the first falsy value or the last value if none were found.

The rules above are similar to OR. The difference is that AND returns the first falsy value while OR returns the first truthy one.

Examples:

// if the first operand is truthy,

// AND returns the second operand:

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

// if the first operand is falsy,

// AND returns it. The second operand is ignored

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0

We can also pass several values in a row. See how the first falsy one is returned:

alert( 1 && 2 && null && 3 ); // null

When all values are truthy, the last value is returned:

alert( 1 && 2 && 3 ); // 3, the last one

**Precedence of AND && is higher than OR ||**

The precedence of AND && operator is higher than OR ||.

So the code a && b || c && d is essentially the same as if the &&expressions were in parentheses: (a && b) || (c && d).

Just like OR, the AND && operator can sometimes replace if.

For instance:

let x = 1;

(x > 0) && alert( 'Greater than zero!' );

The action in the right part of && would execute only if the evaluation reaches it. That is, only if (x > 0) is true.

So we basically have an analogue for:

let x = 1;

if (x > 0) {

alert( 'Greater than zero!' );

}

The variant with && appears shorter. But if is more obvious and tends to be a little bit more readable.

So we recommend using every construct for its purpose: use if if we want if and use && if we want AND.

## [! (NOT)](http://javascript.info/logical-operators" \l "not)

The boolean NOT operator is represented with an exclamation sign !.

The syntax is pretty simple:

result = !value;

The operator accepts a single argument and does the following:

1. Converts the operand to boolean type: true/false.
2. Returns the inverse value.

For instance:

alert( !true ); // false

alert( !0 ); // true

A double NOT !! is sometimes used for converting a value to boolean type:

alert( !!"non-empty string" ); // true

alert( !!null ); // false

That is, the first NOT converts the value to boolean and returns the inverse, and the second NOT inverses it again. In the end, we have a plain value-to-boolean conversion.

There’s a little more verbose way to do the same thing – a built-in Booleanfunction:

alert( Boolean("non-empty string") ); // true

alert( Boolean(null) ); // false

The precedence of NOT ! is the highest of all logical operators, so it always executes first, before && or ||.

Loops

We covered 3 types of loops:

* while – The condition is checked before each iteration.
* do..while – The condition is checked after each iteration.
* for (;;) – The condition is checked before each iteration, additional settings available.

To make an “infinite” loop, usually the while(true) construct is used. Such a loop, just like any other, can be stopped with the break directive.

If we don’t want to do anything in the current iteration and would like to forward to the next one, we can use the continue directive.

break/continue support labels before the loop. A label is the only way for break/continue to escape a nested loop to go to an outer one.

Functions

A function declaration looks like this:

function name(parameters, delimited, by, comma) {

/\* code \*/

}

* Values passed to a function as parameters are copied to its local variables.
* A function may access outer variables. But it works only from inside out. The code outside of the function doesn’t see its local variables.
* A function can return a value. If it doesn’t, then its result is undefined.

To make the code clean and easy to understand, it’s recommended to use mainly local variables and parameters in the function, not outer variables.

It is always easier to understand a function which gets parameters, works with them and returns a result than a function which gets no parameters, but modifies outer variables as a side-effect.

Function naming:

* A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.
* A function is an action, so function names are usually verbal.
* There exist many well-known function prefixes like create…, show…, get…, check… and so on. Use them to hint what a function does.

Functions are the main building blocks of scripts. Now we’ve covered the basics, so we actually can start creating and using them. But that’s only the beginning of the path. We are going to return to them many times, going more deeply into their advanced features.

Function expressions

* Functions are values. They can be assigned, copied or declared in any place of the code.
* If the function is declared as a separate statement in the main code flow, that’s called a “Function Declaration”.
* If the function is created as a part of an expression, it’s called a “Function Expression”.
* Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
* Function Expressions are created when the execution flow reaches them.

In most cases when we need to declare a function, a Function Declaration is preferable, because it is visible prior to the declaration itself. That gives us more flexibility in code organization, and is usually more readable.

So we should use a Function Expression only when a Function Declaration is not fit for the task. We’ve seen a couple of examples of that in this chapter, and will see more in the future.

Arrow functions

6th November 2019

# Arrow functions, the basics

There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.

It’s called “arrow functions”, because it looks like this:

let func = (arg1, arg2, ...argN) => expression

…This creates a function func that accepts arguments arg1..argN, then evaluates the expression on the right side with their use and returns its result.

In other words, it’s the shorter version of:

let func = function(arg1, arg2, ...argN) {

return expression;

};

Let’s see a concrete example:

let sum = (a, b) => a + b;

/\* This arrow function is a shorter form of:

let sum = function(a, b) {

return a + b;

};

\*/

alert( sum(1, 2) ); // 3

As you can, see (a, b) => a + b means a function that accepts two arguments named a and b. Upon the execution, it evaluates the expression a + b and returns the result.

* If we have only one argument, then parentheses around parameters can be omitted, making that even shorter.

For example:

let double = n => n \* 2;

// roughly the same as: let double = function(n) { return n \* 2 }

alert( double(3) ); // 6

* If there are no arguments, parentheses will be empty (but they should be present):
* let sayHi = () => alert("Hello!");

sayHi();

Arrow functions can be used in the same way as Function Expressions.

For instance, to dynamically create a function:

let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?

() => alert('Hello') :

() => alert("Greetings!");

welcome(); // ok now

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure.

They are very convenient for simple one-line actions, when we’re just too lazy to write many words.

## [Multiline arrow functions](http://javascript.info/arrow-functions-basics" \l "multiline-arrow-functions)

The examples above took arguments from the left of => and evaluated the right-side expression with them.

Sometimes we need something a little bit more complex, like multiple expressions or statements. It is also possible, but we should enclose them in curly braces. Then use a normal return within them.

Like this:

let sum = (a, b) => { // the curly brace opens a multiline function

let result = a + b;

return result; // if we use curly braces, then we need an explicit "return"

};

alert( sum(1, 2) ); // 3

**More to come**

Here we praised arrow functions for brevity. But that’s not all!

Arrow functions have other interesting features.

To study them in-depth, we first need to get to know some other aspects of JavaScript, so we’ll return to arrow functions later in the chapter [Arrow functions revisited](http://javascript.info/arrow-functions).

For now, we can already use arrow functions for one-line actions and callbacks.

## [Summary](http://javascript.info/arrow-functions-basics" \l "summary)

Arrow functions are handy for one-liners. They come in two flavors:

1. Without curly braces: (...args) => expression – the right side is an expression: the function evaluates it and returns the result.
2. With curly braces: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit returnto return something.

JavaScript specials – a short BRIEFING

This chapter briefly recaps the features of JavaScript that we’ve learned by now, paying special attention to subtle moments.