**An Introduction to JavaScript**

**[What is JavaScript?](https://javascript.info/intro" \l "what-is-javascript)**

*JavaScript* was initially created to *“make webpages alive”*.

The programs in this language are called *scripts*. They can be written right in the HTML and execute automatically as the page loads.

Scripts are provided and executed as a plain text. They don’t need a special preparation or a compilation to run.

In this aspect, JavaScript is very different from another language called [Java](http://en.wikipedia.org/wiki/Java).

**Why JavaScript?**

When JavaScript was created, it initially had another name: “LiveScript”. But Java language was very popular at that time, so it was decided that positioning a new language as a “younger brother” of Java would help.

But as it evolved, JavaScript became a fully independent language, with its own specification called [ECMAScript](http://en.wikipedia.org/wiki/ECMAScript), and now it has no relation to Java at all.

At present, JavaScript can execute not only in the browser, but also on the server, or actually on any device where there exists a special program called [the JavaScript engine](https://en.wikipedia.org/wiki/JavaScript_engine).

The browser has an embedded engine, sometimes it’s also called a “JavaScript virtual machine”.

Different engines have different “codenames”, for example:

* [V8](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) – in Chrome and Opera.
* [SpiderMonkey](https://en.wikipedia.org/wiki/SpiderMonkey) – in Firefox.
* …There are other codenames like “Trident”, “Chakra” for different versions of IE, “ChakraCore” for Microsoft Edge, “Nitro” and “SquirrelFish” for Safari etc.

The terms above are good to remember, because they are used in developer articles on the internet. We’ll use them too. For instance, if “a feature X is supported by V8”, then it probably works in Chrome and Opera.

**How engines work?**

Engines are complicated. But the basics are easy.

1. The engine (embedded if it’s a browser) reads (“parses”) the script.
2. Then it converts (“compiles”) the script to the machine language.
3. And then the machine code runs, pretty fast.

The engine applies optimizations on every stage of the process. It even watches the compiled script as it runs, analyzes the data that flows through it and applies optimizations to the machine code based on that knowledge. At the end, scripts are quite fast.

**[What can in-browser JavaScript do?](https://javascript.info/intro" \l "what-can-in-browser-javascript-do)**

The modern JavaScript is a “safe” programming language. It does not provide low-level access to memory or CPU, because it was initially created for browsers which do not require it.

The capabilities greatly depend on the environment that runs JavaScript. For instance, [Node.JS](https://wikipedia.org/wiki/Node.js) supports functions that allow JavaScript to read/write arbitrary files, perform network requests etc.

In-browser JavaScript can do everything related to webpage manipulation, interaction with the user and the webserver.

For instance, in-browser JavaScript is able to:

* Add new HTML to the page, change the existing content, modify styles.
* React to user actions, run on mouse clicks, pointer movements, key presses.
* Send requests over the network to remote servers, download and upload files (so-called [AJAX](https://en.wikipedia.org/wiki/Ajax_(programming)) and [COMET](https://en.wikipedia.org/wiki/Comet_(programming)) technologies).
* Get and set cookies, ask questions to the visitor, show messages.
* Remember the data on the client-side (“local storage”).

**[What CAN’T in-browser JavaScript do?](https://javascript.info/intro" \l "what-can-t-in-browser-javascript-do)**

JavaScript’s abilities in the browser are limited for the sake of the user’s safety. The aim is to prevent an evil webpage from accessing private information or harming the user’s data.

The examples of such restrictions are:

* JavaScript on a webpage may not read/write arbitrary files on the hard disk, copy them or execute programs. It has no direct access to OS system functions.

Modern browsers allow it to work with files, but the access is limited and only provided if the user does certain actions, like “dropping” a file into a browser window or selecting it via an <input> tag.

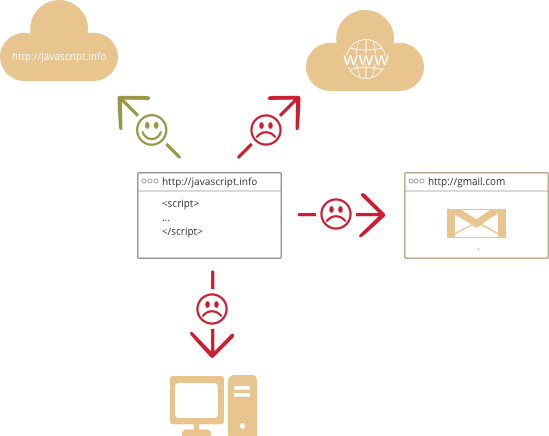
There are ways to interact with camera/microphone and other devices, but they require a user’s explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the [NSA](https://en.wikipedia.org/wiki/National_Security_Agency).

* Different tabs/windows generally do not know about each other. Sometimes they do, for example when one window uses JavaScript to open the other one. But even in this case, JavaScript from one page may not access the other if they come from different sites (from a different domain, protocol or port).

This is called the “Same Origin Policy”. To work around that, *both pages* must contain a special JavaScript code that handles data exchange.

The limitation is again for user’s safety. A page from http://anysite.com which a user has opened must not be able to access another browser tab with the URL http://gmail.com and steal information from there.

* JavaScript can easily communicate over the net to the server where the current page came from. But its ability to receive data from other sites/domains is crippled. Though possible, it requires explicit agreement (expressed in HTTP headers) from the remote side. Once again, that’s safety limitations.



Such limits do not exist if JavaScript is used outside of the browser, for example on a server. Modern browsers also allow installing plugin/extensions which may get extended permissions.

**[What makes JavaScript unique?](https://javascript.info/intro" \l "what-makes-javascript-unique)**

There are at least *three* great things about JavaScript:

* Full integration with HTML/CSS.
* Simple things done simply.
* Supported by all major browsers and enabled by default.

Combined, these three things exist only in JavaScript and no other browser technology.

That’s what makes JavaScript unique. That’s why it’s the most widespread tool to create browser interfaces.

While planning to learn a new technology, it’s beneficial to check its perspectives. So let’s move on to the modern trends that include new languages and browser abilities.

**[Languages “over” JavaScript](https://javascript.info/intro" \l "languages-over-javascript)**

The syntax of JavaScript does not suit everyone’s needs. Different people want different features.

That’s to be expected, because projects and requirements are different for everyone.

So recently a plethora of new languages appeared, which are *transpiled* (converted) to JavaScript before they run in the browser.

Modern tools make the transpilation very fast and transparent, actually allowing developers to code in another language and autoconverting it “under the hood”.

Examples of such languages:

* [CoffeeScript](http://coffeescript.org/) is a “syntactic sugar” for JavaScript, it introduces shorter syntax, allowing to write more precise and clear code. Usually Ruby devs like it.
* [TypeScript](http://www.typescriptlang.org/) is concentrated on adding “strict data typing”, to simplify development and support of complex systems. It is developed by Microsoft.
* [Dart](https://www.dartlang.org/) is a standalone language that has its own engine that runs in non-browser environments (like mobile apps). It was initially offered by Google as a replacement for JavaScript, but as of now, browsers require it to be transpiled to JavaScript just like the ones above.

There are more. Of course even if we use one of those languages, we should also know JavaScript, to really understand what we’re doing.

**[Summary](https://javascript.info/intro" \l "summary)**

* JavaScript was initially created as a browser-only language, but now it is used in many other environments as well.
* At this moment, JavaScript has a unique position as the most widely-adopted browser language with full integration with HTML/CSS.
* There are many languages that get “transpiled” to JavaScript and provide certain features. It is recommended to take a look at them, at least briefly, after mastering JavaScript.

**Code editors**

A code editor is the place where programmers spend most of their time.

There are two archetypes: IDE and lightweight editors. Many people feel comfortable choosing one tool of each type.

**[IDE](https://javascript.info/code-editors" \l "ide)**

The term [IDE](https://en.wikipedia.org/wiki/Integrated_development_environment) (Integrated Development Environment) means a powerful editor with many features that usually operates on a “whole project”. As the name suggests, that’s not just an editor, but a full-scale “development environment”.

An IDE loads the project (can be many files), allows navigation between files, provides autocompletion based on the whole project (not just the open file), integrates with a version management system (like [git](https://git-scm.com/)), a testing environment and other “project-level” stuff.

If you haven’t considered selecting an IDE yet, look at the following variants:

* [WebStorm](http://www.jetbrains.com/webstorm/) for frontend development and other editors of the same company if you need additional languages.
* Visual Studio is fine if you’re a .NET developer, and a free version is available ([Visual Studio Community](https://www.visualstudio.com/vs/community/))
* [Netbeans](http://netbeans.org/).

All of the IDEs except Visual Studio are available on Windows, MacOs and Linux. Visual Studio doesn’t work on Linux.

Most IDEs are paid, but have a trial period. Their cost is usually negligible compared to a qualified developer’s salary, so just choose the best one for you.

**[Lightweight editors](https://javascript.info/code-editors" \l "lightweight-editors)**

“Lightweight editors” are not as powerful as IDEs, but they’re fast, elegant and simple.

They are mainly used to instantly open and edit a file.

The main difference between a “lightweight editor” and an “IDE” is that an IDE works on a project-level, so it loads much more data on start, analyzes the project structure if needed and so on. A lightweight editor is much faster if we need only one file.

In practice, lightweight editors may have a lot of plugins including directory-level syntax analyzers and autocompleters, so there’s no strict border between a lightweight editor and an IDE.

The following options deserve your attention:

* [Visual Studio Code](https://code.visualstudio.com/) (cross-platform, free).
* [Atom](https://atom.io/) (cross-platform, free).
* [Sublime Text](http://www.sublimetext.com/) (cross-platform, shareware).
* [Notepad++](https://notepad-plus-plus.org/) (Windows, free).
* [Vim](http://www.vim.org/) and [Emacs](https://www.gnu.org/software/emacs/) are also cool, if you know how to use them.

**Developer console**

Code is prone to errors. You are quite likely to make errors… Oh, what am I talking about? You are *absolutely* going to make errors, at least if you’re a human, not a [robot](https://en.wikipedia.org/wiki/Bender_(Futurama)).

But in the browser, a user doesn’t see the errors by default. So, if something goes wrong in the script, we won’t see what’s broken and can’t fix it.

To see errors and get a lot of other useful information about scripts, browsers have embedded “developer tools”.

Most often developers lean towards Chrome or Firefox for development, because those browsers have the best developer tools. Other browsers also provide developer tools, sometimes with special features, but are usually playing “catch-up” to Chrome or Firefox. So most people have a “favorite” browser and switch to others if a problem is browser-specific.

Developer tools are really powerful, there are many features. To start, we’ll learn how to open them, look at errors and run JavaScript commands.

**[Google Chrome](https://javascript.info/devtools" \l "google-chrome)**

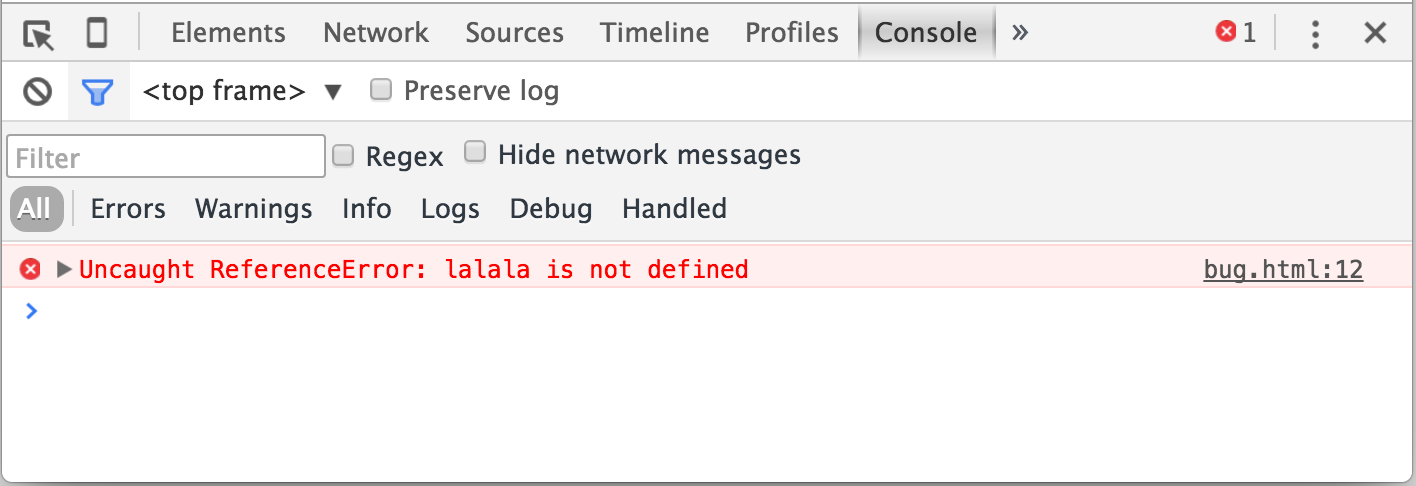
Open the page [bug.html](https://javascript.info/article/devtools/bug.html).

There’s an error in the JavaScript code on it. It’s hidden from a regular visitor’s eyes, so let’s open developer tools to see it.

Press F12 or, if you’re on Mac, then Cmd+Opt+J.

The developer tools will open on the Console tab by default.

It looks somewhat like this:



The exact look of developer tools depends on your version of Chrome. It changes from time to time, but should be similar.

* Here we can see the red-colored error message. In this case the script contains an unknown “lalala” command.
* On the right, there is a clickable link to the source bug.html:12 with the line number where the error has occurred.

Below the error message there is a blue > symbol. It marks a “command line” where we can type JavaScript commands and press Enter to run them (Shift+Enter to input multi-line commands).

Now we can see errors and that’s enough for the start. We’ll be back to developer tools later and cover debugging more in-depth in the chapter [Debugging in Chrome](https://javascript.info/debugging-chrome).

**[Firefox, Edge and others](https://javascript.info/devtools" \l "firefox-edge-and-others)**

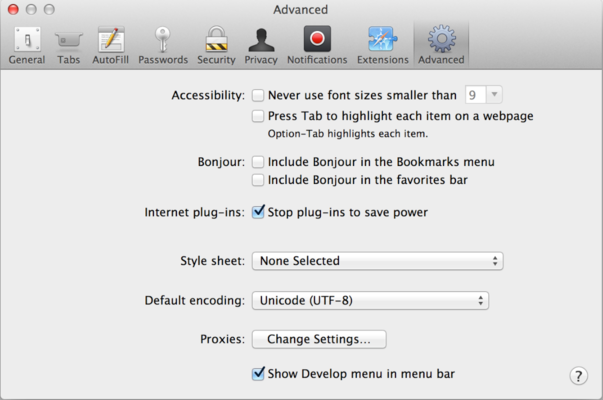
Most other browsers use F12 to open developer tools.

The look & feel of them is quite similar. Once you know how to use one of them (you can start with Chrome), you can easily switch to another.

**[Safari](https://javascript.info/devtools" \l "safari)**

Safari (Mac browser, not supported by Windows/Linux) is a little bit special here. We need to enable the “Develop menu” first.

Open Preferences and go to “Advanced” pane. There’s a checkbox at the bottom:



Now Cmd+Opt+C can toggle the console. Also note that the new top menu item named “Develop” has appeared. It has many commands and options.

[**Summary**](https://javascript.info/devtools#summary)

* Developer tools allow us to see errors, run commands, examine variables and much more.
* They can be opened with F12 for most browsers under Windows. Chrome for Mac needs Cmd+Opt+J, Safari: Cmd+Opt+C(need to enable first).

Apps:

<!DOCTYPE HTML>

<html>

<body>

<p>Before the script...</p>

<script>

alert( 'Hello, world!' );

</script>

<p>...After the script.</p>

</body>

</html>

# Code structure

The first thing to study is the building blocks of the code.

## [Statements](https://javascript.info/structure" \l "statements)

Statements are syntax constructs and commands that perform actions.

We’ve already seen a statement alert('Hello, world!'), which shows the message.

We can have as many statements in the code as we want. Another statement can be separated with a semicolon.

For example, here we split the message into two:

alert('Hello'); alert('World');

Usually each statement is written on a separate line – thus the code becomes more readable:

alert('Hello');

alert('World');

## [Semicolons](https://javascript.info/structure" \l "semicolon)

A semicolon may be omitted in most cases when a line break exists.

This would also work:

alert('Hello')

alert('World')

Here JavaScript interprets the line break as an “implicit” semicolon. That’s also called an [automatic semicolon insertion](https://tc39.github.io/ecma262/#sec-automatic-semicolon-insertion).

**In most cases a newline implies a semicolon. But “in most cases” does not mean “always”!**

There are cases when a newline does not mean a semicolon, for example:

alert(3 +

1

+ 2);

The code outputs 6, because JavaScript does not insert semicolons here. It is intuitively obvious that if the line ends with a plus "+", then it is an “incomplete expression”, no semicolon required. And in this case that works as intended.

**But there are situations where JavaScript “fails” to assume a semicolon where it is really needed.**

Errors which occur in such cases are quite hard to find and fix.

**An example of an error**

If you’re curious to see a concrete example of such an error, check this code out:

[1, 2].forEach(alert)

No need to think about the meaning of the brackets [] and forEach yet. We’ll study them later, for now it does not matter. Let’s just remember the result: it shows 1, then 2.

Now let’s add an alert before the code and not finish it with a semicolon:

alert("There will be an error")

[1, 2].forEach(alert)

Now if we run it, only the first alert is shown, and then we have an error!

But everything is fine again if we add a semicolon after alert:

alert("All fine now");

[1, 2].forEach(alert)

Now we have the “All fine now” message and then 1 and 2.

The error in the no-semicolon variant occurs because JavaScript does not imply a semicolon before square brackets [...].

So, because the semicolon is not auto-inserted, the code in the first example is treated as a single statement. That’s how the engine sees it:

alert("There will be an error")[1, 2].forEach(alert)

But it should be two separate statements, not a single one. Such a merging in this case is just wrong, hence the error. There are other situations when such a thing happens.

It’s recommended to put semicolons between statements even if they are separated by newlines. This rule is widely adopted by the community. Let’s note once again – it is possible to leave out semicolons most of the time. But it’s safer – especially for a beginner – to use them.

## [Comments](https://javascript.info/structure" \l "comments)

As time goes on, the program becomes more and more complex. It becomes necessary to add comments which describe what happens and why.

Comments can be put into any place of the script. They don’t affect the execution, because the engine simply ignores them.

**One-line comments start with two forward slash characters //.**

The rest of the line is a comment. It may occupy a full line of its own or follow a statement.

Like here:

// This comment occupies a line of its own

alert('Hello');

alert('World'); // This comment follows the statement

**Multiline comments start with a forward slash and an asterisk /\* and end with an asterisk and a forward slash \*/.**

Like this:

/\* An example with two messages.

This is a multiline comment.

\*/

alert('Hello');

alert('World');

The content of comments is ignored, so if we put code inside /\* … \*/ it won’t execute.

Sometimes it comes in handy to temporarily disable a part of code:

/\* Commenting out the code

alert('Hello');

\*/

alert('World');

**Use hotkeys!**

In most editors a line of code can be commented out by Ctrl+/ hotkey for a single-line comment and something like Ctrl+Shift+/ – for multiline comments (select a piece of code and press the hotkey). For Mac try Cmd instead of Ctrl.

**Nested comments are not supported!**

There may not be /\*...\*/ inside another /\*...\*/.

Such code will die with an error:

/\*

/\* nested comment ?!? \*/

\*/

alert( 'World' );

Please, don’t hesitate to comment your code.

Comments increase the overall code footprint, but that’s not a problem at all. There are many tools which minify the code before publishing to the production server. They remove comments, so they don’t appear in the working scripts. Therefore comments do not have any negative effects on production at all.

Further in the tutorial, there will be a chapter [Coding style](https://javascript.info/coding-style) that also explains how to write better comments.

**The modern mode, "use strict"**

For a long time JavaScript was evolving without compatibility issues. New features were added to the language, but the old functionality did not change.

That had the benefit of never breaking existing code. But the downside was that any mistake or an imperfect decision made by JavaScript creators got stuck in the language forever.

It had been so until 2009 when ECMAScript 5 (ES5) appeared. It added new features to the language and modified some of the existing ones. To keep the old code working, most modifications are off by default. One needs to enable them explicitly with a special directive "use strict".

**[“use strict”](https://javascript.info/strict-mode" \l "use-strict)**

The directive looks like a string: "use strict" or 'use strict'. When it is located on the top of the script, then the whole script works the “modern” way.

For example

"use strict";

// this code works the modern way

...

We will learn functions (a way to group commands) soon.

Looking ahead let’s just note that "use strict" can be put at the start of a function (most kinds of functions) instead of the whole script. Then strict mode is enabled in that function only. But usually people use it for the whole script.

**Ensure that “use strict” is at the top**

Please make sure that "use strict" is on the top of the script, otherwise the strict mode may not be enabled.

There is no strict mode here:

alert("some code");

// "use strict" below is ignored, must be on the top

"use strict";

// strict mode is not activated

Only comments may appear above "use strict".

**There’s no way to cancel use strict**

There is no directive "no use strict" or alike, that would return the old behavior.

Once we enter the strict mode, there’s no return.

**[Always “use strict”](https://javascript.info/strict-mode" \l "always-use-strict)**

The differences of "use strict" versus the “default” mode are still to be covered.

In the next chapters, as we learn language features, we’ll make notes about the differences of the strict mode. Luckily, there are not so many. And they actually make our life better.

At this point in time 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 as we study.
2. The strict mode is enabled by "use strict" at the top. Also there are several language features like “classes” and “modules” that enable strict mode automatically.
3. The strict mode is supported by all modern browsers.
4. It’s always recommended to start scripts with "use strict". All examples in this tutorial assume so, unless (very rarely) specified otherwise.

# Variables

Most of the time, a JavaScript application needs to work with information. Here are 2 examples:

1. An online-shop – the information might include goods being sold and a shopping cart.
2. A chat application – the information might include users, messages, and much more.

Variables are used to store this information.

## [A variable](https://javascript.info/variables" \l "a-variable)

A [variable](https://en.wikipedia.org/wiki/Variable_(computer_science)) is a “named storage” for data. We can use variables to store goodies, visitors and other data.

To create a variable in JavaScript, we need to use the let keyword.

The statement below creates (in other words: declares or defines) a variable with the name “message”:

let message;

Now we can put some data into it by using the assignment operator =:

let message;

message = 'Hello'; // store the string

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

let message;

message = 'Hello!';

alert(message); // shows the variable content

To be concise we can merge the variable declaration and assignment into a single line:

let message = 'Hello!'; // define the variable and assign the value

alert(message); // Hello!

We can also declare multiple variables in one line:

let user = 'John', age = 25, message = 'Hello';

That might seem shorter, but it’s not recommended. For the sake of better readability, please use a single line per variable.

The multiline variant is a bit longer, but easier to read:

let user = 'John';

let age = 25;

let message = 'Hello';

Some people also write many variables like that:

let user = 'John',

age = 25,

message = 'Hello';

…Or even in the “comma-first” style:

let user = 'John'

, age = 25

, message = 'Hello';

Technically, all these variants do the same. So, it’s a matter of personal taste and aesthetics.

**var instead of let**

In older scripts you may also find another keyword: var instead of let:

var message = 'Hello';

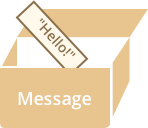
The var keyword is almost the same as let. It also declares a variable, but in a slightly different, “old-school” fashion.

There are subtle differences between let and var, but they do not matter for us yet. We’ll cover them in detail later, in the chapter [The old "var"](https://javascript.info/var).

## [A real-life analogy](https://javascript.info/variables" \l "a-real-life-analogy)

We can easily grasp the concept of a “variable” if we imagine it as a “box” for data, with a uniquely-named sticker on it.

For instance, the variable message can be imagined as a box labeled "message" with the value "Hello!" in it:



We can put any value into the box.

Also we can change it. The value can be changed as many times as needed:

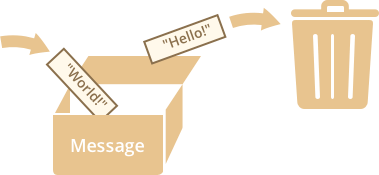
let message;

message = 'Hello!';

message = 'World!'; // value changed

alert(message);

When the value is changed, the old data is removed from the variable:



We can also declare two variables and copy data from one into the other.

let hello = 'Hello world!';

let message;

// copy 'Hello world' from hello into message

message = hello;

// now two variables hold the same data

alert(hello); // Hello world!

alert(message); // Hello world!

**Functional languages**

It may be interesting to know that there also exist [functional](https://en.wikipedia.org/wiki/Functional_programming) programming languages that forbid changing a variable value. For example, [Scala](http://www.scala-lang.org/) or [Erlang](http://www.erlang.org/).

In such languages, once the value is stored “in the box”, it’s there forever. If we need to store something else, the language forces us to create a new box (declare a new variable). We can’t reuse the old one.

Though it may seem a little bit odd at first sight, these languages are quite capable of serious development. More than that, there are areas like parallel computations where this limitation confers certain benefits. Studying such a language (even if not planning to use it soon) is recommended to broaden the mind.

## [Variable naming](https://javascript.info/variables" \l "variable-naming)

There are two limitations for a variable name in JavaScript:

1. The name must contain only letters, digits, symbols $ and \_.
2. The first character must not be a digit.

Valid names, for instance:

let userName;

let test123;

When the name contains multiple words, [camelCase](https://en.wikipedia.org/wiki/CamelCase) is commonly used. That is: words go one after another, each word starts with a capital letter: myVeryLongName.

What’s interesting – the dollar sign '$' and the underscore '\_' can also be used in names. They are regular symbols, just like letters, without any special meaning.

These names are valid:

let $ = 1; // declared a variable with the name "$"

let \_ = 2; // and now a variable with the name "\_"

alert($ + \_); // 3

Examples of incorrect variable names:

let 1a; // cannot start with a digit

let my-name; // a hyphen '-' is not allowed in the name

**Case matters**

Variables named apple and AppLE – are two different variables.

**Non-english letters are allowed, but not recommended**

It is possible to use any language, including cyrillic letters or even hieroglyphs, like this:

let имя = '...';

let 我 = '...';

Technically, there is no error here, such names are allowed, but there is an international tradition to use English in variable names. Even if we’re writing a small script, it may have a long life ahead. People from other countries may need to read it some time.

**Reserved names**

There is a list of reserved words, which cannot be used as variable names, because they are used by the language itself.

For example, words let, class, return, function are reserved.

The code below gives a syntax error:

let let = 5; // can't name a variable "let", error!

let return = 5; // also can't name it "return", error!

**An assignment without use strict**

Normally, we need to define a variable before using it. But in the old times, it was technically possible to create a variable by a mere assignment of the value, without let. This still works now if we don’t put use strict. The behavior is kept for compatibility with old scripts.

// note: no "use strict" in this example

num = 5; // the variable "num" is created if didn't exist

alert(num); // 5

That’s a bad practice, it gives an error in the strict mode:

"use strict";

num = 5; // error: num is not defined

## [Constants](https://javascript.info/variables" \l "constants)

To declare a constant (unchanging) variable, one can use const instead of let:

const myBirthday = '18.04.1982';

Variables declared using const are called “constants”. They cannot be changed. An attempt to do it would cause an error:

const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant!

When a programmer is sure that the variable should never change, he can use const to guarantee it, and also to clearly show that fact to everyone.

### [Uppercase constants](https://javascript.info/variables" \l "uppercase-constants)

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution.

Such constants are named using capital letters and underscores.

Like this:

const COLOR\_RED = "#F00";

const COLOR\_GREEN = "#0F0";

const COLOR\_BLUE = "#00F";

const COLOR\_ORANGE = "#FF7F00";

// ...when we need to pick a color

let color = COLOR\_ORANGE;

alert(color); // #FF7F00

Benefits:

* COLOR\_ORANGE is much easier to remember than "#FF7F00".
* It is much easier to mistype in "#FF7F00" than in COLOR\_ORANGE.
* When reading the code, COLOR\_ORANGE is much more meaningful than #FF7F00.

When should we use capitals for a constant, and when should we name them normally? Let’s make that clear.

Being a “constant” just means that the value never changes. But there are constants that are known prior to execution (like a hexadecimal value for red), and there are those that are calculated in run-time, during the execution, but do not change after the assignment.

For instance:

const pageLoadTime = /\* time taken by a webpage to load \*/;

The value of pageLoadTime is not known prior to the page load, so it’s named normally. But it’s still a constant, because it doesn’t change after assignment.

In other words, capital-named constants are only used as aliases for “hard-coded” values.

## [Name things right](https://javascript.info/variables" \l "name-things-right)

Talking about variables, there’s one more extremely important thing.

Please name the variables sensibly. Take time to think if needed.

Variable naming is one of the most important and complex skills in programming. A quick glance at variable names can reveal which code is written by a beginner and which by an experienced developer.

In a real project, most of the time is spent on modifying and extending the existing code base, rather than writing something completely separate from scratch. And when we return to the code after some time of doing something else, it’s much easier to find information that is well-labeled. Or, in other words, when the variables have good names.

Please spend some time thinking about the right name for a variable before declaring it. This will repay you a lot.

Some good-to-follow rules are:

* Use human-readable names like userName or shoppingCart.
* Stay away from abbreviations or short names like a, b, c, unless you really know what you’re doing.
* Make the name maximally descriptive and concise. Examples of bad names are data and value. Such a name says nothing. It is only ok to use them if it’s exceptionally obvious from the context which data or value is meant.
* Agree on terms within your team and in your own mind. If a site visitor is called a “user” then we should name related variables like currentUser or newUser, but not currentVisitor or a newManInTown.

Sounds simple? Indeed it is, but creating good descriptive-and-concise names in practice is not. Go for it.

**Reuse or create?**

And the last note. There are some lazy programmers who, instead of declaring a new variable, tend to reuse the existing ones.

As a result, the variable is like a box where people throw different things without changing the sticker. What is inside it now? Who knows… We need to come closer and check.

Such a programmer saves a little bit on variable declaration, but loses ten times more on debugging the code.

An extra variable is good, not evil.

Modern JavaScript minifiers and browsers optimize code well enough, so it won’t create performance issues. Using different variables for different values can even help the engine to optimize.

## [Summary](https://javascript.info/variables#summary)

We can declare variables to store data. That can be done using var or let or const.

* let – is a modern variable declaration. The code must be in strict mode to use let in Chrome (V8).
* 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"](https://javascript.info/var), just in case you need them.
* const – is like let, but the value of the variable can’t be changed.

# Data types

A variable in JavaScript can contain any data. A variable can at one moment be a string and later receive a numeric value:

// no error

let message = "hello";

message = 123456;

Programming languages that allow such things are called “dynamically typed”, meaning that there are data types, but variables are not bound to any of them.

There are seven basic data types in JavaScript. Here we’ll study the basics, and in the next chapters we’ll talk about each of them in detail.

## [A number](https://javascript.info/types" \l "a-number)

let n = 123;

n = 12.345;

The number type serves both for integer and floating point numbers.

There are many operations for numbers, e.g. multiplication \*, division /, addition +, subtraction - and so on.

Besides regular numbers, there are so-called “special numeric values” which also belong to that type: Infinity, -Infinityand NaN.

* Infinity represents the mathematical [Infinity](https://en.wikipedia.org/wiki/Infinity) ∞. It is a special value that’s greater than any number.

We can get it as a result of division by zero:

alert( 1 / 0 ); // Infinity

Or just mention it in the code directly:

alert( Infinity ); // Infinity

* NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

alert( "not a number" / 2 ); // NaN, such division is erroneous

NaN is sticky. Any further operation on NaN would give NaN:

alert( "not a number" / 2 + 5 ); // NaN

So, if there’s NaN somewhere in a mathematical expression, it propagates to the whole result.

**Mathematical operations are safe**

Doing maths is safe in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc.

The script will never stop with a fatal error (“die”). At worst we’ll get NaN as the result.

Special numeric values formally belong to the “number” type. Of course they are not numbers in a common sense of this word.

We’ll see more about working with numbers in the chapter [Numbers](https://javascript.info/number).

## [A string](https://javascript.info/types" \l "a-string)

A string in JavaScript must be quoted.

let str = "Hello";

let str2 = 'Single quotes are ok too';

let phrase = `can embed ${str}`;

In JavaScript, there are 3 types of quotes.

1. Double quotes: "Hello".
2. Single quotes: 'Hello'.
3. Backticks: `Hello`.

Double and single quotes are “simple” quotes. There’s no difference between them in JavaScript.

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}, for example:

let name = "John";

// embed a variable

alert( `Hello, ${name}!` ); // Hello, John!

// embed an expression

alert( `the result is ${1 + 2}` ); // the result is 3

The expression inside ${…} is evaluated and the result becomes a part of the string. We can put anything there: a variable like name or an arithmetical expression like 1 + 2 or something more complex.

Please note that this can only be done in backticks. Other quotes do not allow such embedding!

alert( "the result is ${1 + 2}" ); // the result is ${1 + 2} (double quotes do nothing)

We’ll cover strings more thoroughly in the chapter [Strings](https://javascript.info/string).

**There is no character type.**

In some languages, there is a special “character” type for a single character. For example, in the C language and in Java it is char.

In JavaScript, there is no such type. There’s only one type: string. A string may consist of only one character or many of them.

## [A boolean (logical type)](https://javascript.info/types" \l "a-boolean-logical-type)

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

For instance:

let nameFieldChecked = true; // yes, name field is checked

let ageFieldChecked = false; // no, age field is not checked

Boolean values also come as a result of comparisons:

let isGreater = 4 > 1;

alert( isGreater ); // true (the comparison result is "yes")

We’ll cover booleans more deeply later in the chapter [Logical operators](https://javascript.info/logical-operators).

## [The “null” value](https://javascript.info/types" \l "the-null-value)

The special null value does not belong to any type of those described above.

It forms a separate type of its own, which contains only the null value:

let age = null;

In JavaScript null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

It’s just a special value which has the sense of “nothing”, “empty” or “value unknown”.

The code above states that the age is unknown or empty for some reason.

## [The “undefined” value](https://javascript.info/types" \l "the-undefined-value)

The special value undefined stands apart. It makes a type of its own, just like null.

The meaning of undefined is “value is not assigned”.

If a variable is declared, but not assigned, then its value is exactly undefined:

let x;

alert(x); // shows "undefined"

Technically, it is possible to assign undefined to any variable:

let x = 123;

x = undefined;

alert(x); // "undefined"

…But it’s not recommended to do that. Normally, we use null to write an “empty” or an “unknown” value into the variable, and undefined is only used for checks, to see if the variable is assigned or similar.

## [Objects and Symbols](https://javascript.info/types" \l "objects-and-symbols)

The object type is special.

All other types are called “primitive”, because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities. We’ll deal with them later in the chapter [Objects](https://javascript.info/object) after we know enough about primitives.

The symbol type is used to create unique identifiers for objects. We have to mention it here for completeness, but it’s better to study them after objects.

## [The typeof operator](https://javascript.info/types" \l "type-typeof)

The typeof operator returns the type of the argument. It’s useful when we want to process values of different types differently, or just want to make a quick check.

It supports two forms of syntax:

1. As an operator: typeof x.
2. Function style: typeof(x).

In other words, it works both with parentheses or without them. The result is the same.

The call to typeof x returns a string with the type name:

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)

The last three lines may need additional explanations:

1. Math is a built-in object that provides mathematical operations. We will learn it in the chapter [Numbers](https://javascript.info/number). Here it serves just as an example of an object.
2. The result of typeof null is "object". That’s wrong. It is an officially recognized error in typeof, kept for compatibility. Of course, null is not an object. It is a special value with a separate type of its own. So, again, that’s an error in the language.
3. The result of typeof alert is "function", because alert is a function of the language. We’ll study functions in the next chapters, and we’ll see that there’s no special “function” type in the language. Functions belong to the object type. But typeof treats them differently. Formally, it’s incorrect, but very convenient in practice.

## [Summary](https://javascript.info/types#summary)

There are 7 basic 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 the variable.

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

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

## [Tasks](https://javascript.info/types#tasks)

### [String quotes](https://javascript.info/types" \l "string-quotes)

importance: 5

What is the output of the script?

let name = "Ilya";

alert( `hello ${1}` ); // ?

alert( `hello ${"name"}` ); // ?

alert( `hello ${name}` ); // ?

# Type Conversions

Most of the time, operators and functions automatically convert a value to the right type. That’s called “type conversion”.

For example, alert automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

There are also cases when we need to explicitly convert a value to put things right.

**Not talking about objects yet**

In this chapter we don’t cover objects yet. Here we study primitives first. Later, after we learn objects, we’ll see how object conversion works in the chapter [Object to primitive conversion](https://javascript.info/object-toprimitive).

## [ToString](https://javascript.info/type-conversions" \l "tostring)

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value.

We can also use a call String(value) function for that:

let value = true;

alert(typeof value); // boolean

value = String(value); // now value is a string "true"

alert(typeof value); // string

String conversion is mostly obvious. A false becomes "false", null becomes "null" etc.

## [ToNumber](https://javascript.info/type-conversions" \l "tonumber)

Numeric conversion happens in mathematical functions and expressions automatically.

For example, when division / is applied to non-numbers:

alert( "6" / "2" ); // 3, strings are converted to numbers

We can use a Number(value) function to explicitly convert a value:

let str = "123";

alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number

Explicit conversion is usually required when we read a value from a string-based source like a text form, but we expect a number to be entered.

If the string is not a valid number, the result of such conversion is NaN, for instance:

let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, conversion failed

Numeric conversion rules:

| **Value** | **Becomes…** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1 and 0 |
| string | Whitespaces from the start and the end are removed. Then, if the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN. |

Examples:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (error reading a number at "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

Please note that null and undefined behave differently here: null becomes a zero, while undefined becomes NaN.

**Addition ‘+’ concatenates strings**

Almost all mathematical operations convert values to numbers. With a notable exception of the addition +. If one of the added values is a string, then another one is also converted to a string.

Then it concatenates (joins) them:

alert( 1 + '2' ); // '12' (string to the right)

alert( '1' + 2 ); // '12' (string to the left)

That only happens when one of the arguments is a string. Otherwise, values are converted to numbers.

## [ToBoolean](https://javascript.info/type-conversions" \l "toboolean)

Boolean conversion is the simplest one.

It happens in logical operations (later we’ll meet condition tests and other kinds of them), but also can be performed manually with the call of Boolean(value).

The conversion rule:

* Values that are intuitively “empty”, like 0, an empty string, null, undefined and NaN become false.
* Other values become true.

For instance:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false

**Please note: the string with zero "0" is true**

Some languages (namely PHP) treat "0" as false. But in JavaScript a non-empty string is always true.

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)

## [Summary](https://javascript.info/type-conversions#summary)

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

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

**ToNumber** – 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. |

**ToBoolean** – Occurs in logical operations, or 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 are not covered here, we’ll return to them later in the chapter [Object to primitive conversion](https://javascript.info/object-toprimitive) that is devoted exclusively to objects, after we learn more basic things about JavaScript.

## [Tasks](https://javascript.info/type-conversions#tasks)

### [Type conversions](https://javascript.info/type-conversions" \l "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\n" + 5

" -9\n" - 5

null + 1

undefined + 1

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

solution

**Operators**

Many operators are known to us from school. They are addition +, a multiplication \*, a subtraction - and so on.

In this chapter we concentrate on aspects that are not covered by school arithmetic.

**[Terms: “unary”, “binary”, “operand”](https://javascript.info/operators" \l "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 negation - reverses the sign of the 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 the binary form as well:
* let x = 1, y = 3;

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

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

**[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 the operands is a string, then 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, then convert the other one 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. 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 + 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 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 is shorter.

A need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, then 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 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 the 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) (you don’t need to remember this, but note that unary operators are higher than corresponding binary ones):

| **Precedence** | **Name** | **Sign** |
| --- | --- | --- |
| … | … | … |
| 16 | unary plus | + |
| 16 | unary negation | - |
| 14 | multiplication | \* |
| 14 | division | / |
| 13 | addition | + |
| 13 | subtraction | - |
| … | … | … |
| 3 | assignment | = |
| … | … | … |

As we can see, the “unary plus” has a priority of 16, which is 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

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 then assigned to the variables on the left: c, b and a. At the end, all 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 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 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 subtract 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 its 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 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 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 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 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 counter 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 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, then 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 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++;

**[Bitwise operators](https://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 ( ^ )

1 0 1

0 1 1

0 0 0

1 1 0

* NOT ( ~ )
* 0 1
* 1 0
* LEFT SHIFT ( << )

110

100

* RIGHT SHIFT ( >> )
* 011
* ZERO-FILL RIGHT SHIFT ( >>> )

These operators are used very rarely. To understand them, we should delve into low-level number representation, and it would not be optimal to do that right now. Especially because 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 in MDN. It would be more practical to do that when a real need arises.

**[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:

X=x+1 or x+=1 x\*=1 x/= 1 x-=1

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)

**[Comma](https://javascript.info/operators" \l "comma)**

The comma operator , is one of most rare and 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 the result of only 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 then the number after the comma 7 is not processed anyhow, so it’s ignored.

Why do we need such an operator which throws away everything except the last part?

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 mention them. But usually they don’t improve the code readability, so we should think well before writing like that.

[**Tasks**](https://javascript.info/operators#tasks)

**[The postfix and prefix forms](https://javascript.info/operators" \l "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](https://javascript.info/operators" \l "assignment-result)**

importance: 3

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

let a = 2;

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

# Comparisons

Many comparison operators we know from maths:

* Greater/less than: a > b, a < b.
* Greater/less than or equals: a >= b, a <= b.
* Equality check is written as a == b (please note the double equation sign =. A single symbol a = b would mean an assignment).
* Not equals. In maths the notation is ≠, in JavaScript it’s written as an assignment with an exclamation sign before it: a != b.

## [Boolean is the result](https://javascript.info/comparison" \l "boolean-is-the-result)

Just as all other operators, a comparison returns a value. The value is of the boolean type.

* true – means “yes”, “correct” or “the truth”.
* false – means “no”, “wrong” or “a lie”.

For example:

alert( 2 > 1 ); // true (correct)

alert( 2 == 1 ); // false (wrong)

alert( 2 != 1 ); // true (correct)

A comparison result can be assigned to a variable, just like any value:

let result = 5 > 4; // assign the result of the comparison

alert( result ); // true

## [String comparison](https://javascript.info/comparison" \l "string-comparison)

To see which string is greater than the other, the so-called “dictionary” or “lexicographical” order is used.

In other words, strings are compared letter-by-letter.

For example:

alert( 'Z' > 'A' ); // true

alert( 'Glow' > 'Glee' ); // true

alert( 'Bee' > 'Be' ); // true

The algorithm to compare two strings is simple:

1. Compare first characters of both strings.
2. If the first one is greater(or less), then the first string is greater(or less) than the second. We’re done.
3. Otherwise if first characters are equal, compare the second characters the same way.
4. Repeat until the end of any string.
5. If both strings ended simultaneously, then they are equal. Otherwise the longer string is greater.

In the example above, the comparison 'Z' > 'A' gets the result at the first step.

Strings "Glow" and "Glee" are compared character-by-character:

1. G is the same as G.
2. l is the same as l.
3. o is greater than e. Stop here. The first string is greater.

**Not a real dictionary, but Unicode order**

The comparison algorithm given above is roughly equivalent to the one used in book dictionaries or phone books. But it’s not exactly the same.

For instance, case matters. A capital letter "A" is not equal to the lowercase "a". Which one is greater? Actually, the lowercase "a" is. Why? Because the lowercase character has a greater index in the internal encoding table (Unicode). We’ll get back to specific details and consequences in the chapter [Strings](https://javascript.info/string).

## [Comparison of different types](https://javascript.info/comparison" \l "comparison-of-different-types)

When compared values belong to different types, they are converted to numbers.

For example:

alert( '2' > 1 ); // true, string '2' becomes a number 2

alert( '01' == 1 ); // true, string '01' becomes a number 1

For boolean values, true becomes 1 and false becomes 0, that’s why:

alert( true == 1 ); // true

alert( false == 0 ); // true

**A funny consequence**

It is possible that at the same time:

* Two values are equal.
* One of them is true as a boolean and the other one is false as a boolean.

For example:

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

alert(a == b); // true!

From JavaScript’s standpoint that’s quite normal. An equality check converts using the numeric conversion (hence "0"becomes 0), while Boolean conversion uses another set of rules.

## [Strict equality](https://javascript.info/comparison" \l "strict-equality)

A regular equality check == has a problem. It cannot differ 0 from false:

alert( 0 == false ); // true

The same thing with an empty string:

alert( '' == false ); // true

That’s because operands of different types are converted to a number by the equality operator ==. An empty string, just like false, becomes a zero.

What to do if we’d like to differentiate 0 from false?

**A strict equality operator === checks the equality without type conversion.**

In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them.

Let’s try it:

alert( 0 === false ); // false, because the types are different

There also exists a “strict non-equality” operator !==, as an analogy for !=.

The strict equality check operator is a bit longer to write, but makes it obvious what’s going on and leaves less space for errors.

## [Comparison with null and undefined](https://javascript.info/comparison" \l "comparison-with-null-and-undefined)

Let’s see more edge cases.

There’s a non-intuitive behavior when null or undefined are compared with other values.

**For a strict equality check ===**

These values are different, because each of them belongs to a separate type of its own.

alert( null === undefined ); // false

**For a non-strict check ==**

There’s a special rule. These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value.

alert( null == undefined ); // true

**For maths and other comparisons < > <= >=**

Values null/undefined are converted to a number: null becomes 0, while undefined becomes NaN.

Now let’s see funny things that happen when we apply those rules. And, what’s more important, how to not fall into a trap with these features.

### [Strange result: null vs 0](https://javascript.info/comparison" \l "strange-result-null-vs-0)

Let’s compare null with a zero:

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

Yeah, mathematically that’s strange. The last result states that "null is greater than or equal to zero". Then one of the comparisons above must be correct, but they are both false.

The reason is that an equality check == and comparisons > < >= <= work differently. Comparisons convert null to a number, hence treat it as 0. That’s why (3) null >= 0 is true and (1) null > 0 is false.

On the other hand, the equality check == for undefined and null works by the rule, without any conversions. They equal each other and don’t equal anything else. That’s why (2) null == 0 is false.

### [An incomparable undefined](https://javascript.info/comparison" \l "an-incomparable-undefined)

The value undefined shouldn’t participate in comparisons at all:

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

Why does it dislike a zero so much? Always false!

We’ve got these results because:

* Comparisons (1) and (2) return false because undefined gets converted to NaN. And NaN is a special numeric value which returns false for all comparisons.
* The equality check (3) returns false, because undefined only equals null and no other value.

### [Evade problems](https://javascript.info/comparison" \l "evade-problems)

Why did we observe these examples? Should we remember these peculiarities all the time? Well, not really. Actually, these tricky things will gradually become familiar over time, but there’s a solid way to evade any problems with them.

Just treat any comparison with undefined/null except the strict equality === with exceptional care.

Don’t use comparisons >= > < <= with a variable which may be null/undefined, unless you are really sure what you’re doing. If a variable can have such values, then check for them separately.

## [Summary](https://javascript.info/comparison#summary)

* Comparison operators return a logical 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).
* 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. Making a separate check for null/undefined is a good idea.

# Interaction: alert, prompt, confirm

This part of the tutorial aims to cover JavaScript “as is”, without environment-specific tweaks.

But still we use a browser as the demo environment. So we should know at least a few user-interface functions. In this chapter we’ll get familiar with the browser functions alert, prompt and confirm.

## [alert](https://javascript.info/alert-prompt-confirm" \l "alert)

Syntax:

alert(message);

This shows a message and pauses the script execution until the user presses “OK”.

For example:

alert("Hello");

The mini-window with the message is called a modal window. The word “modal” means that the visitor can’t interact with the rest of the page, press other buttons etc, until they have dealt with the window. In this case – until they press “OK”.

## [prompt](https://javascript.info/alert-prompt-confirm" \l "prompt)

Function prompt accepts two arguments:

result = prompt(title[, default]);

It shows a modal window with a text message, an input field for the visitor and buttons OK/CANCEL.

**title**

The text to show to the visitor.

**default**

An optional second parameter, the initial value for the input field.

The visitor may type something in the prompt input field and press OK. Or they can cancel the input by pressing the CANCEL button or hitting the Esc key.

The call to prompt returns the text from the field or null if the input was canceled.

For instance:

let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!

**IE: always supply a default**

The second parameter is optional. But if we don’t supply it, Internet Explorer would insert the text "undefined" into the prompt.

Run this code in Internet Explorer to see that:

let test = prompt("Test");

So, to look good in IE, it’s recommended to always provide the second argument:

let test = prompt("Test", ''); // <-- for IE

## [confirm](https://javascript.info/alert-prompt-confirm" \l "confirm)

The syntax:

result = confirm(question);

Function confirm shows a modal window with a question and two buttons: OK and CANCEL.

The result is true if OK is pressed and false otherwise.

For example:

let isBoss = confirm("Are you the boss?");

alert( isBoss ); // true if OK is pressed

## [Summary](https://javascript.info/alert-prompt-confirm#summary)

We covered 3 browser-specific functions to interact with the visitor:

**alert**

shows a message.

**prompt**

shows a message asking the user to input text. It returns the text or, if CANCEL or Esc is clicked, all browsers return 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 the script execution and don’t allow the visitor to interact with the rest of the page until the message 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.

## [Tasks](https://javascript.info/alert-prompt-confirm#tasks)

### [A simple page](https://javascript.info/alert-prompt-confirm" \l "a-simple-page)

importance: 4

Create a web-page that asks for a name and outputs it.

# Conditional operators: if, '?'

Sometimes we need to perform different actions based on a condition.

There is the if statement for that and also the conditional (ternary) operator for conditional evaluation which we will be referring as the “question mark” operator ? for simplicity.

## [The “if” statement](https://javascript.info/ifelse" \l "the-if-statement)

The if statement gets a condition, evaluates it and, if the result is true, executes the code.

For example:

let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year == 2015) alert( 'You are right!' );

In the example above, the condition is a simple equality check: year == 2015, but it can be much more complex.

If there is more than one statement to be executed, we have to wrap our code block inside curly braces:

if (year == 2015) {

alert( "That's correct!" );

alert( "You're so smart!" );

}

It is recommended to wrap your code block with curly braces {} every time with if, even if there is only one statement. That improves readability.

## [Boolean conversion](https://javascript.info/ifelse" \l "boolean-conversion)

The if (…) statement evaluates the expression in parentheses and converts it to the boolean type.

Let’s recall the conversion rules from the chapter [Type Conversions](https://javascript.info/type-conversions):

* A number 0, an empty string "", null, undefined and NaN become false. Because of that they are called “falsy” values.
* Other values become true, so they are called “truthy”.

So, the code under this condition would never execute:

if (0) { // 0 is falsy

...

}

…And inside this condition – always works:

if (1) { // 1 is truthy

...

}

We can also pass a pre-evaluated boolean value to if, like here:

let cond = (year == 2015); // equality evaluates to true or false

if (cond) {

...

}

## [The “else” clause](https://javascript.info/ifelse" \l "the-else-clause)

The if statement may contain an optional “else” block. It executes when the condition is wrong.

For example:

let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year == 2015) {

alert( 'You guessed it right!' );

} else {

alert( 'How can you be so wrong?' ); // any value except 2015

}

## [Several conditions: “else if”](https://javascript.info/ifelse" \l "several-conditions-else-if)

Sometimes we’d like to test several variants of a condition. There is an else if clause for that.

For example:

let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year < 2015) {

alert( 'Too early...' );

} else if (year > 2015) {

alert( 'Too late' );

} else {

alert( 'Exactly!' );

}

In the code above JavaScript first checks year < 2015. If it is falsy it then goes to the next condition year > 2015, and otherwise shows the last alert.

There can be more else if blocks. The ending else is optional.

## [Ternary operator ‘?’](https://javascript.info/ifelse" \l "ternary-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 “ternary” or “question mark” operator lets us do that shorter and simpler.

The operator is represented by a question mark ?. The formal term “ternary” means that 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 parentheses around age > 18. The question mark operator has a low precedence. It executes after the comparison >, so that’ll do the same:

// 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 it’s recommended to use them.

**Please note:**

In the example above it’s possible to evade the question mark operator, because the comparison by itself returns true/false:

// the same

let accessAllowed = age > 18;

## [Multiple ‘?’](https://javascript.info/ifelse" \l "multiple)

A sequence of question mark ? operators allows returning 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 – returns 'Hi, baby!', otherwise – goes after the colon ":" and checks for age < 18.
3. If that’s true – returns 'Hello!', otherwise – goes after the next colon ":" and checks for age < 100.
4. If that’s true – returns 'Greetings!', otherwise – goes after the last colon ":" and returns 'What an unusual age!'.

The same logic 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!';

}

## [Non-traditional use of ‘?’](https://javascript.info/ifelse" \l "non-traditional-use-of)

Sometimes the question mark ? is used as a replacement for if:

let company = prompt('Which company created JavaScript?', '');

(company == 'Netscape') ?

alert('Right!') : alert('Wrong.');

Depending on the condition company == 'Netscape', either the first or the second part after ? gets executed and shows the alert.

We don’t assign a result to a variable here. The idea is to execute different code depending on the condition.

**It is not recommended to use the question mark operator in this way.**

The notation seems to be shorter than if, which appeals to some programmers. But it is less readable.

Here is the same code with if for comparison:

let company = prompt('Which company created JavaScript?', '');

if (company == 'Netscape') {

alert('Right!');

} else {

alert('Wrong.');

}

Our eyes scan the code vertically. The constructs which span several lines are easier to understand than a long horizontal instruction set.

The idea of a question mark ? is to return one or another value depending on the condition. Please use it for exactly that. There is if to execute different branches of the code.

## [Tasks](https://javascript.info/ifelse#tasks)

### [if (a string with zero)](https://javascript.info/ifelse" \l "if-a-string-with-zero)

importance: 5

Will alert be shown?

if ("0") {

alert( 'Hello' );

}

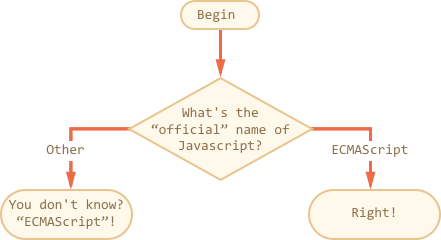
solution

### [The name of JavaScript](https://javascript.info/ifelse" \l "the-name-of-javascript)

importance: 2

Using the if..else construct, write the code which asks: ‘What is the “official” name of JavaScript?’

If the visitor enters “ECMAScript”, then output “Right!”, otherwise – output: “Didn’t know? ECMAScript!”



[Demo in new window](https://en.js.cx/task/check-standard/ifelse_task2/)

solution

### [Show the sign](https://javascript.info/ifelse" \l "show-the-sign)

importance: 2

Using if..else, write the code which gets a number via prompt and then shows in alert:

* 1, if the value is greater than zero,
* -1, if less than zero,
* 0, if equals zero.

In this task we assume that the input is always a number.

[Demo in new window](https://en.js.cx/task/sign/if_sign/)

solution

### [Check the login](https://javascript.info/ifelse" \l "check-the-login)

importance: 3

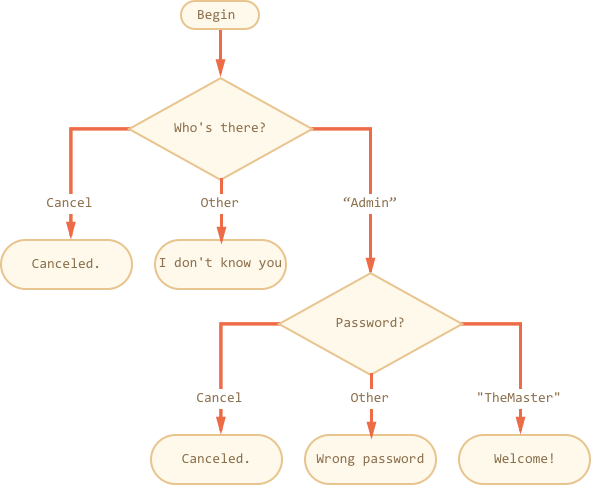
Write the code which asks for a login with prompt.

If the visitor enters "Admin", then prompt for a password, if the input is an empty line or Esc – show “Canceled.”, if it’s another string – then show “I don’t know you”.

The password is checked as follows:

* If it equals “TheMaster”, then show “Welcome!”,
* Another string – show “Wrong password”,
* For an empty string or cancelled input, show “Canceled.”

The schema:



Please use nested if blocks. Mind the overall readability of the code.

Hint: passing an empty input to a prompt returns an empty string ''. Pressing ESC during a prompt returns null.

[Run the demo](https://javascript.info/ifelse)

solution

### [Rewrite 'if' into '?'](https://javascript.info/ifelse" \l "rewrite-if-into)

importance: 5

Rewrite this if using the ternary operator '?':

if (a + b < 4) {

result = 'Below';

} else {

result = 'Over';

}

solution

### [Rewrite 'if..else' into '?'](https://javascript.info/ifelse" \l "rewrite-if-else-into)

importance: 5

Rewrite if..else using multiple ternary operators '?'.

For readability, it’s recommended to split the code into multiple lines.

let message;

if (login == 'Employee') {

message = 'Hello';

} else if (login == 'Director') {

message = 'Greetings';

} else if (login == '') {

message = 'No login';

} else {

message = '';

}

# 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. The result can also be of any type.

Let’s see the details.

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

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

result = a || b;

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

In JavaScript the operator is a little bit more tricky and 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 boolean, then it’s converted to boolean for the evaluation.

For instance, a number 1 is treated as true, a 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 correct.

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 seeks the first truthy value](https://javascript.info/logical-operators" \l "or-seeks-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:

* Evaluate operands from left to right.
* For each operand, convert it to boolean. If the result is true, then stop and return the original value of that operand.
* If all other operands have been assessed (i.e. all were false), return 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 such 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)

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

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

Imagine we have several variables, which can either contain the data or be null/undefined. And we need to choose the first one with data.

We can use OR || for that:

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 then "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. The 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.

If we run the example below, x would not get assigned:

let x;

true || (x = 1);

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

…And if the first argument is false, then OR goes on and evaluates the second one thus running the assignment:

let x;

false || (x = 1);

alert(x); // 1

An assignment is a simple case, other side effects can be involved.

As we can see, such a use case is a "shorter way to do if". The first operand is converted to boolean and if it’s false then 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 that can be handy.

## [&& (AND)](https://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( 'Time is 12:30' );

}

Just as for 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 seeks the first falsy value](https://javascript.info/logical-operators" \l "and-seeks-the-first-falsy-value)

Given multiple AND’ed values:

result = value1 && value2 && value3;

The AND && operator does the following:

* Evaluate operands from left to right.
* For each operand, convert it to a boolean. If the result is false, stop and return the original value of that operand.
* If all other operands have been assessed (i.e. all were truthy), return 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

**AND && executes before OR ||**

The precedence of the AND && operator is higher than OR ||, so it executes before OR.

In the code below 1 && 0 is calculated first:

alert( 5 || 1 && 0 ); // 5

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 to be shorter. But if is more obvious and tends to be a little bit more readable.

So it is recommended to use every construct for its purpose. Use if if we want if. And use && if we want AND.

## [! (NOT)](https://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 an 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. At 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 Boolean function:

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

alert( Boolean(null) ); // false

## [Tasks](https://javascript.info/logical-operators#tasks)

### [What's the result of OR?](https://javascript.info/logical-operators" \l "what-s-the-result-of-or)

importance: 5

What the code below is going to output?

alert( null || 2 || undefined );

solution

### [What's the result of OR'ed alerts?](https://javascript.info/logical-operators" \l "what-s-the-result-of-or-ed-alerts)

importance: 3

What the code below will output?

alert( alert(1) || 2 || alert(3) );

solution

### [What is the result of AND?](https://javascript.info/logical-operators" \l "what-is-the-result-of-and)

importance: 5

What this code is going to show?

alert( 1 && null && 2 );

solution

### [What is the result of AND'ed alerts?](https://javascript.info/logical-operators" \l "what-is-the-result-of-and-ed-alerts)

importance: 3

What will this code show?

alert( alert(1) && alert(2) );

solution

### [The result of OR AND OR](https://javascript.info/logical-operators" \l "the-result-of-or-and-or)

importance: 5

What will be the result?

alert( null || 2 && 3 || 4 );

solution

### [Check the range between](https://javascript.info/logical-operators" \l "check-the-range-between)

importance: 3

Write an “if” condition to check that age is between 14 and 90 inclusively.

“Inclusively” means that age can reach the edges 14 or 90.

solution

### [Check the range outside](https://javascript.info/logical-operators" \l "check-the-range-outside)

importance: 3

Write an if condition to check that age is NOT between 14 and 90 inclusively.

Create two variants: the first one using NOT !, the second one – without it.

solution

### [A question about "if"](https://javascript.info/logical-operators" \l "a-question-about-if)

importance: 5

Which of these alerts are going to execute?

What will be the results of the expressions inside if(...)?

if (-1 || 0) alert( 'first' );

if (-1 && 0) alert( 'second' );

if (null || -1 && 1) alert( 'third' );

# Loops: while and for

We often have a need to perform similar actions many times in a row.

For example, when we need to output goods from a list one after another. Or just run the same code for each number from 1 to 10.

Loops are a way to repeat the same part of code multiple times.

## [The “while” loop](https://javascript.info/while-for" \l "the-while-loop)

The while loop has the following syntax:

while (condition) {

// code

// so-called "loop body"

}

While the condition is true, the code from the loop body is executed.

For instance, the loop below outputs i while i < 3:

let i = 0;

while (i < 3) { // shows 0, then 1, then 2

alert( i );

i++;

}

A single execution of the loop body is called an iteration. The loop in the example above makes three iterations.

If there were no i++ in the example above, the loop would repeat (in theory) forever. In practice, the browser provides ways to stop such loops, and for server-side JavaScript we can kill the process.

Any expression or a variable can be a loop condition, not just a comparison. They are evaluated and converted to a boolean by while.

For instance, the shorter way to write while (i != 0) could be while (i):

let i = 3;

while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops

alert( i );

i--;

}

**Brackets are not required for a single-line body**

If the loop body has a single statement, we can omit the brackets {…}:

let i = 3;

while (i) alert(i--);

## [The “do…while” loop](https://javascript.info/while-for" \l "the-do-while-loop)

The condition check can be moved below the loop body using the do..while syntax:

do {

// loop body

} while (condition);

The loop will first execute the body, then check the condition and, while it’s truthy, execute it again and again.

For example:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

This form of syntax is rarely used except when you want the body of the loop to execute **at least once** regardless of the condition being truthy. Usually, the other form is preferred: while(…) {…}.

## [The “for” loop](https://javascript.info/while-for" \l "the-for-loop)

The for loop is the most often used one.

It looks like this:

for (begin; condition; step) {

// ... loop body ...

}

Let’s learn the meaning of these parts by example. The loop below runs alert(i) for i from 0 up to (but not including) 3:

for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2

alert(i);

}

Let’s examine the for statement part by part:

| **part** |  |  |
| --- | --- | --- |
| begin | i = 0 | Executes once upon entering the loop. |
| condition | i < 3 | Checked before every loop iteration, if fails the loop stops. |
| step | i++ | Executes after the body on each iteration, but before the condition check. |
| body | alert(i) | Runs again and again while the condition is truthy |

The general loop algorithm works like this:

Run begin

→ (if condition → run body and run step)

→ (if condition → run body and run step)

→ (if condition → run body and run step)

→ ...

If you are new to loops, then maybe it would help if you go back to the example and reproduce how it runs step-by-step on a piece of paper.

Here’s what exactly happens in our case:

// for (let i = 0; i < 3; i++) alert(i)

// run begin

let i = 0

// if condition → run body and run step

if (i < 3) { alert(i); i++ }

// if condition → run body and run step

if (i < 3) { alert(i); i++ }

// if condition → run body and run step

if (i < 3) { alert(i); i++ }

// ...finish, because now i == 3

**Inline variable declaration**

Here the “counter” variable i is declared right in the loop. That’s called an “inline” variable declaration. Such variables are visible only inside the loop.

for (let i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // error, no such variable

Instead of defining a variable, we can use an existing one:

let i = 0;

for (i = 0; i < 3; i++) { // use an existing variable

alert(i); // 0, 1, 2

}

alert(i); // 3, visible, because declared outside of the loop

### [Skipping parts](https://javascript.info/while-for" \l "skipping-parts)

Any part of for can be skipped.

For example, we can omit begin if we don’t need to do anything at the loop start.

Like here:

let i = 0; // we have i already declared and assigned

for (; i < 3; i++) { // no need for "begin"

alert( i ); // 0, 1, 2

}

We can also remove the step part:

let i = 0;

for (; i < 3;) {

alert( i++ );

}

The loop became identical to while (i < 3).

We can actually remove everything, thus creating an infinite loop:

for (;;) {

// repeats without limits

}

Please note that the two for semicolons ; must be present, otherwise it would be a syntax error.

## [Breaking the loop](https://javascript.info/while-for" \l "breaking-the-loop)

Normally the loop exits when the condition becomes falsy.

But we can force the exit at any moment. There’s a special break directive for that.

For example, the loop below asks the user for a series of numbers, but “breaks” when no number is entered:

let sum = 0;

while (true) {

let value = +prompt("Enter a number", '');

if (!value) break; // (\*)

sum += value;

}

alert( 'Sum: ' + sum );

The break directive is activated at the line (\*) if the user enters an empty line or cancels the input. It stops the loop immediately, passing the control to the first line after the loop. Namely, alert.

The combination “infinite loop + break as needed” is great for situations when the condition must be checked not in the beginning/end of the loop, but in the middle, or even in several places of the body.

## [Continue to the next iteration](https://javascript.info/while-for" \l "continue)

The continue directive is a “lighter version” of break. It doesn’t stop the whole loop. Instead it stops the current iteration and forces the loop to start a new one (if the condition allows).

We can use it if we’re done on the current iteration and would like to move on to the next.

The loop below uses continue to output only odd values:

for (let i = 0; i < 10; i++) {

// if true, skip the remaining part of the body

if (i % 2 == 0) continue;

alert(i); // 1, then 3, 5, 7, 9

}

For even values of i the continue directive stops body execution, passing the control to the next iteration of for (with the next number). So the alert is only called for odd values.

**The directive continue helps to decrease nesting level**

A loop that shows odd values could look like this:

for (let i = 0; i < 10; i++) {

if (i % 2) {

alert( i );

}

}

From a technical point of view it’s identical to the example above. Surely, we can just wrap the code in the if block instead of continue.

But as a side-effect we got one more nesting level (the alert call inside the curly braces). If the code inside if is longer than a few lines, that may decrease the overall readability.

**No break/continue to the right side of ‘?’**

Please note that syntax constructs that are not expressions cannot be used with the ternary operator ?. In particular, directives such as break/continue are disallowed there.

For example, if we take this code:

if (i > 5) {

alert(i);

} else {

continue;

}

…And rewrite it using a question mark:

(i > 5) ? alert(i) : continue; // continue not allowed here

…Then it stops working. The code like this will give a syntax error:

That’s just another reason not to use a question mark operator ? instead of if.

## [Labels for break/continue](https://javascript.info/while-for" \l "labels-for-break-continue)

Sometimes we need to break out from multiple nested loops at once.

For example, in the code below we loop over i and j prompting for coordinates (i, j) from (0,0) to (3,3):

for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Value at coords (${i},${j})`, '');

// what if I want to exit from here to Done (below)?

}

}

alert('Done!');

We need a way to stop the process if the user cancels the input.

The ordinary break after input would only break the inner loop. That’s not sufficient. Labels come to the rescue.

A label is an identifier with a colon before a loop:

labelName: for (...) {

...

}

The break <labelName> statement in the loop breaks out to the label.

Like here:

outer: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Value at coords (${i},${j})`, '');

// if an empty string or canceled, then break out of both loops

if (!input) break outer; // (\*)

// do something with the value...

}

}

alert('Done!');

In the code above break outer looks upwards for the label named outer and breaks out of that loop.

So the control goes straight from (\*) to alert('Done!').

We can also move the label onto a separate line:

outer:

for (let i = 0; i < 3; i++) { ... }

The continue directive can also be used with a label. In this case the execution jumps to the next iteration of the labeled loop.

**Labels are not a “goto”**

Labels do not allow us to jump into an arbitrary place of code.

For example, it is impossible to do this:

break label; // jumps to label? No.

label: for (...)

The call to a break/continue is only possible from inside the loop, and the label must be somewhere upwards from the directive.

## [Summary](https://javascript.info/while-for#summary)

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 on the current iteration and would like to forward to the next one, the continue directive does it.

break/continue support labels before the loop. A label is the only way for break/continue to escape the nesting and go to the outer loop.

## [Tasks](https://javascript.info/while-for#tasks)

### [Last loop value](https://javascript.info/while-for" \l "last-loop-value)

importance: 3

What is the last value alerted by this code? Why?

let i = 3;

while (i) {

alert( i-- );

}

solution

### [Which values shows the while?](https://javascript.info/while-for" \l "which-values-shows-the-while)

importance: 4

For every loop, write down which values it shows, in your opinion. And then compare with the answer.

Both loops alert same values or not?

1. The prefix form ++i:
2. let i = 0;

while (++i < 5) alert( i );

1. The postfix form i++
2. let i = 0;

while (i++ < 5) alert( i );

solution

### [Which values get shown by the "for" loop?](https://javascript.info/while-for" \l "which-values-get-shown-by-the-for-loop)

importance: 4

For each loop write down which values it is going to show. Then compare with the answer.

Both loops alert same values or not?

1. The postfix form:

for (let i = 0; i < 5; i++) alert( i );

1. The prefix form:

for (let i = 0; i < 5; ++i) alert( i );

solution

### [Output even numbers in the loop](https://javascript.info/while-for" \l "output-even-numbers-in-the-loop)

importance: 5

Use the for loop to output even numbers from 2 to 10.

[Run the demo](https://javascript.info/while-for)

solution

### [Replace "for" with "while"](https://javascript.info/while-for" \l "replace-for-with-while)

importance: 5

Rewrite the code changing the for loop to while without altering its behavior (the output should stay same).

for (let i = 0; i < 3; i++) {

alert( `number ${i}!` );

}

solution

### [Repeat until the input is correct](https://javascript.info/while-for" \l "repeat-until-the-input-is-correct)

importance: 5

Write a loop which prompts for a number greater than 100. If the visitor enters another number – ask him to input again.

The loop must ask for a number until either the visitor enters a number greater than 100 or cancels the input/enters an empty line.

Here we can assume that the visitor only inputs numbers. There’s no need to implement a special handling for a non-numeric input in this task.

[Run the demo](https://javascript.info/while-for)

solution

### [Output prime numbers](https://javascript.info/while-for" \l "output-prime-numbers)

importance: 3

An integer number greater than 1 is called a [prime](https://en.wikipedia.org/wiki/Prime_number) if it cannot be divided without a remainder by anything except 1 and itself.

In other words, n > 1 is a prime if it can’t be evenly divided by anything except 1 and n.

For example, 5 is a prime, because it cannot be divided without a remainder by 2, 3 and 4.

**Write the code which outputs prime numbers in the interval from 2 to n.**

For n = 10 the result will be 2,3,5,7.

P.S. The code should work for any n, not be hard-tuned for any fixed value.

solution

# The "switch" statement

A switch statement can replace multiple if checks.

It gives a more descriptive way to compare a value with multiple variants.

## [The syntax](https://javascript.info/switch" \l "the-syntax)

The switch has one or more case blocks and an optional default.

It looks like this:

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

}

* The value of x is checked for a strict equality to the value from the first case (that is, value1) then to the second (value2) and so on.
* If the equality is found, switch starts to execute the code starting from the corresponding case, until the nearest break(or until the end of switch).
* If no case is matched then the default code is executed (if it exists).

## [An example](https://javascript.info/switch" \l "an-example)

An example of switch (the executed code is highlighted):

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Too small' );

break;

case 4:

alert( 'Exactly!' );

break;

case 5:

alert( 'Too large' );

break;

default:

alert( "I don't know such values" );

}

Here the switch starts to compare a from the first case variant that is 3. The match fails.

Then 4. That’s a match, so the execution starts from case 4 until the nearest break.

**If there is no break then the execution continues with the next case without any checks.**

An example without break:

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Too small' );

case 4:

alert( 'Exactly!' );

case 5:

alert( 'Too big' );

default:

alert( "I don't know such values" );

}

In the example above we’ll see sequential execution of three alerts:

alert( 'Exactly!' );

alert( 'Too big' );

alert( "I don't know such values" );

**Any expression can be a switch/case argument**

Both switch and case allow arbitrary expressions.

For example:

let a = "1";

let b = 0;

switch (+a) {

case b + 1:

alert("this runs, because +a is 1, exactly equals b+1");

break;

default:

alert("this doesn't run");

}

Here +a gives 1, that’s compared with b + 1 in case, and the corresponding code is executed.

## [Grouping of “case”](https://javascript.info/switch" \l "grouping-of-case)

Several variants of case which share the same code can be grouped.

For example, if we want the same code to run for case 3 and case 5:

let a = 2 + 2;

switch (a) {

case 4:

alert('Right!');

break;

case 3: // (\*) grouped two cases

case 5:

case 7:

alert('Wrong!');

alert("Why don't you take a math class?");

break;

default:

alert('The result is strange. Really.');

}

Now both 3 and 5 show the same message.

The ability to “group” cases is a side-effect of how switch/case works without break. Here the execution of case 3 starts from the line (\*) and goes through case 5, because there’s no break.

## [Type matters](https://javascript.info/switch" \l "type-matters)

Let’s emphasize that the equality check is always strict. The values must be of the same type to match.

For example, let’s consider the code:

let arg = prompt("Enter a value?")

switch (arg) {

case '0':

case '1':

alert( 'One or zero' );

break;

case '2':

alert( 'Two' );

break;

case 3:

alert( 'Never executes!' );

break;

default:

alert( 'An unknown value' )

}

1. For 0, 1, the first alert runs.
2. For 2 the second alert runs.
3. But for 3, the result of the prompt is a string "3", which is not strictly equal === to the number 3. So we’ve got a dead code in case 3! The default variant will execute.

## [Tasks](https://javascript.info/switch#tasks)

### [Rewrite the "switch" into an "if"](https://javascript.info/switch" \l "rewrite-the-switch-into-an-if)

importance: 5

Write the code using if..else which would correspond to the following switch:

switch (browser) {

case 'Edge':

alert( "You've got the Edge!" );

break;

case 'Chrome':

case 'Firefox':

case 'Safari':

case 'Opera':

alert( 'Okay we support these browsers too' );

break;

default:

alert( 'We hope that this page looks ok!' );

}

solution

### [Rewrite "if" into "switch"](https://javascript.info/switch" \l "rewrite-if-into-switch)

importance: 4

Rewrite the code below using a single switch statement:

let a = +prompt('a?', '');

if (a == 0) {

alert( 0 );

}

if (a == 1) {

alert( 1 );

}

if (a == 2 || a == 3) {

alert( '2,3' );

}

**Functions**

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like alert(message), prompt(message, default) and confirm(question). But we can create functions of our own as well.

**[Function Declaration](https://javascript.info/function-basics" \l "function-declaration)**

To create a function we can use a *function declaration*.

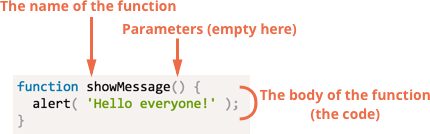
It looks like this:

function showMessage() {

alert( 'Hello everyone!' );

}

The function keyword goes first, then goes the *name of the function*, then a list of *parameters* between the parentheses (empty in the example above) and finally the code of the function, also named “the function body”, between curly braces.



Our new function can be called by its name: showMessage().

For instance:

function showMessage() {

alert( 'Hello everyone!' );

}

showMessage();

showMessage();

The call showMessage() executes the code of the function. Here we will see the message two times.

This example clearly demonstrates one of the main purposes of functions: to avoid code duplication.

If we ever need to change the message or the way it is shown, it’s enough to modify the code in one place: the function which outputs it.

**[Local variables](https://javascript.info/function-basics" \l "local-variables)**

A variable declared inside a function is only visible inside that function.

For example:

function showMessage() {

let message = "Hello, I'm JavaScript!"; // local variable

alert( message );

}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- Error! The variable is local to the function

**[Outer variables](https://javascript.info/function-basics" \l "outer-variables)**

A function can access an outer variable as well, for example:

let userName = 'John';

function showMessage() {

let message = 'Hello, ' + userName;

alert(message);

}

showMessage(); // Hello, John

The function has full access to the outer variable. It can modify it as well.

For instance:

let userName = 'John';

function showMessage() {

userName = "Bob"; // (1) changed the outer variable

let message = 'Hello, ' + userName;

alert(message);

}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function

The outer variable is only used if there’s no local one. So an occasional modification may happen if we forget let.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local userName. The outer one is ignored:

let userName = 'John';

function showMessage() {

let userName = "Bob"; // declare a local variable

let message = 'Hello, ' + userName; // Bob

alert(message);

}

// the function will create and use its own userName

showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable

**Global variables**

Variables declared outside of any function, such as the outer userName in the code above, are called *global*.

Global variables are visible from any function (unless shadowed by locals).

Usually, a function declares all variables specific to its task. Global variables only store project-level data, so when it’s important that these variables are accesible from anywhere. Modern code has few or no globals. Most variables reside in their functions.

**[Parameters](https://javascript.info/function-basics" \l "parameters)**

We can pass arbitrary data to functions using parameters (also called *function arguments*) .

In the example below, the function has two parameters: from and text.

function showMessage(from, text) { // arguments: from, text

alert(from + ': ' + text);

}

showMessage('Ann', 'Hello!'); // Ann: Hello! (\*)

showMessage('Ann', "What's up?"); // Ann: What's up? (\*\*)

When the function is called in lines (\*) and (\*\*), the given values are copied to local variables from and text. Then the function uses them.

Here’s one more example: we have a variable from and pass it to the function. Please note: the function changes from, but the change is not seen outside, because a function always gets a copy of the value:

function showMessage(from, text) {

from = '\*' + from + '\*'; // make "from" look nicer

alert( from + ': ' + text );

}

let from = "Ann";

showMessage(from, "Hello"); // \*Ann\*: Hello

// the value of "from" is the same, the function modified a local copy

alert( from ); // Ann

**[Default values](https://javascript.info/function-basics" \l "default-values)**

If a parameter is not provided, then its value becomes undefined.

For instance, the aforementioned function showMessage(from, text) can be called with a single argument:

showMessage("Ann");

That’s not an error. Such a call would output "Ann: undefined". There’s no text, so it’s assumed that text === undefined.

If we want to use a “default” text in this case, then we can specify it after =:

function showMessage(from, text = "no text given") {

alert( from + ": " + text );

}

showMessage("Ann"); // Ann: no text given

Now if the text parameter is not passed, it will get the value "no text given"

Here "no text given" is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

function showMessage(from, text = anotherFunction()) {

// anotherFunction() only executed if no text given

// its result becomes the value of text

}

**Default parameters old-style**

Old editions of JavaScript did not support default parameters. So there are alternative ways to support them, that you can find mostly in the old scripts.

For instance, an explicit check for being undefined:

function showMessage(from, text) {

if (text === undefined) {

text = 'no text given';

}

alert( from + ": " + text );

}

…Or the || operator:

function showMessage(from, text) {

// if text is falsy then text gets the "default" value

text = text || 'no text given';

...

}

**[Returning a value](https://javascript.info/function-basics" \l "returning-a-value)**

A function can return a value back into the calling code as the result.

The simplest example would be a function that sums two values:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

There may be many occurrences of return in a single function. For instance:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Got a permission from the parents?');

}

}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {

alert( 'Access granted' );

} else {

alert( 'Access denied' );

}

It is possible to use return without a value. That causes the function to exit immediately.

For example:

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Showing you the movie" ); // (\*)

// ...

}

In the code above, if checkAge(age) returns false, then showMovie won’t proceed to the alert.

**A function with an empty return or without it returns undefined**

If a function does not return a value, it is the same as if it returns undefined:

function doNothing() { /\* empty \*/ }

alert( doNothing() === undefined ); // true

An empty return is also the same as return undefined:

function doNothing() {

return;

}

alert( doNothing() === undefined ); // true

**Never add a newline between return and the value**

For a long expression in return, it might be tempting to put it on a separate line, like this:

return

(some + long + expression + or + whatever \* f(a) + f(b))

That doesn’t work, because JavaScript assumes a semicolon after return. That’ll work the same as:

return;

(some + long + expression + or + whatever \* f(a) + f(b))

So, it effectively becomes an empty return. We should put the value on the same line instead.

**[Naming a function](https://javascript.info/function-basics" \l "function-naming)**

Functions are actions. So their name is usually a verb. It should briefly, but as accurately as possible describe what the function does. So that a person who reads the code gets the right clue.

It is a widespread practice to start a function with a verbal prefix which vaguely describes the action. There must be an agreement within the team on the meaning of the prefixes.

For instance, functions that start with "show" usually show something.

Function starting with…

* "get…" – return a value,
* "calc…" – calculate something,
* "create…" – create something,
* "check…" – check something and return a boolean, etc.

Examples of such names:

showMessage(..) // shows a message

getAge(..) // returns the age (gets it somehow)

calcSum(..) // calculates a sum and returns the result

createForm(..) // creates a form (and usually returns it)

checkPermission(..) // checks a permission, returns true/false

With prefixes in place, a glance at a function name gives an understanding what kind of work it does and what kind of value it returns.

**One function – one action**

A function should do exactly what is suggested by its name, no more.

Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

A few examples of breaking this rule:

* getAge – would be bad if it shows an alert with the age (should only get).
* createForm – would be bad if it modifies the document, adding a form to it (should only create it and return).
* checkPermission – would be bad if displays the access granted/denied message (should only perform the check and return the result).

These examples assume common meanings of prefixes. What they mean for you is determined by you and your team. Maybe it’s pretty normal for your code to behave differently. But you should have a firm understanding of what a prefix means, what a prefixed function can and cannot do. All same-prefixed functions should obey the rules. And the team should share the knowledge.

**Ultrashort function names**

Functions that are used *very often* sometimes have ultrashort names.

For example, the [jQuery](http://jquery.com/) framework defines a function $. The [LoDash](http://lodash.com/) library has its core function named \_.

These are exceptions. Generally functions names should be concise, but descriptive.

**[Functions == Comments](https://javascript.info/function-basics" \l "functions-comments)**

Functions should be short and do exactly one thing. If that thing is big, maybe it’s worth it to split the function into a few smaller functions. Sometimes following this rule may not be that easy, but it’s definitely a good thing.

A separate function is not only easier to test and debug – its very existence is a great comment!

For instance, compare the two functions showPrimes(n) below. Each one outputs [prime numbers](https://en.wikipedia.org/wiki/Prime_number) up to n.

The first variant uses a label:

function showPrimes(n) {

nextPrime: for (let i = 2; i < n; i++) {

for (let j = 2; j < i; j++) {

if (i % j == 0) continue nextPrime;

}

alert( i ); // a prime

}

}

The second variant uses an additional function isPrime(n) to test for primality:

function showPrimes(n) {

for (let i = 2; i < n; i++) {

if (!isPrime(i)) continue;

alert(i); // a prime

}

}

function isPrime(n) {

for (let i = 2; i < n; i++) {

if ( n % i == 0) return false;

}

return true;

}

The second variant is easier to understand, isn’t it? Instead of the code piece we see a name of the action (isPrime). Sometimes people refer to such code as *self-describing*.

So, functions can be created even if we don’t intend to reuse them. They structure the code and make it readable.

[**Summary**](https://javascript.info/function-basics#summary)

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.

[**Tasks**](https://javascript.info/function-basics#tasks)

**[Is "else" required?](https://javascript.info/function-basics" \l "is-else-required)**

importance: 4

The following function returns true if the parameter age is greater than 18.

Otherwise it asks for a confirmation and returns its result:

function checkAge(age) {

if (age > 18) {

return true;

} else {

// ...

return confirm('Did parents allow you?');

}

}

Will the function work differently if else is removed?

function checkAge(age) {

if (age > 18) {

return true;

}

// ...

return confirm('Did parents allow you?');

}

Is there any difference in the behavior of these two variants?

solution

**[Rewrite the function using '?' or '||'](https://javascript.info/function-basics" \l "rewrite-the-function-using-or)**

importance: 4

The following function returns true if the parameter age is greater than 18.

Otherwise it asks for a confirmation and returns its result.

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Do you have your parents permission to access this page?');

}

}

Rewrite it, to perform the same, but without if, in a single line.

Make two variants of checkAge:

1. Using a question mark operator ?
2. Using OR ||

solution

**[Function min(a, b)](https://javascript.info/function-basics" \l "function-min-a-b)**

importance: 1

Write a function min(a,b) which returns the least of two numbers a and b.

For instance:

min(2, 5) == 2

min(3, -1) == -1

min(1, 1) == 1

solution

**[Function pow(x,n)](https://javascript.info/function-basics" \l "function-pow-x-n)**

importance: 4

Write a function pow(x,n) that returns x in power n. Or, in other words, multiplies x by itself n times and returns the result.

pow(3, 2) = 3 \* 3 = 9

pow(3, 3) = 3 \* 3 \* 3 = 27

pow(1, 100) = 1 \* 1 \* ...\*1 = 1

Create a web-page that prompts for x and n, and then shows the result of pow(x,n).

[Run the demo](https://javascript.info/function-basics)

P.S. In this task the function should support only natural values of n: integers up from 1.

**Function expressions and arrows**

In JavaScript, a function is not a “magical language structure”, but a special kind of value.

The syntax that we used before is called a *Function Declaration*:

function sayHi() {

alert( "Hello" );

}

There is another syntax for creating a function that is called a *Function Expression*.

It looks like this:

let sayHi = function() {

alert( "Hello" );

};

Here, the function is created and assigned to the variable explicitly, like any other value. No matter how the function is defined, it’s just a value stored in the variable sayHi.

The meaning of these code samples is the same: "create a function and put it into the variable sayHi".

We can even print out that value using alert:

function sayHi() {

alert( "Hello" );

}

alert( sayHi ); // shows the function code

Please note that the last line does not run the function, because there are no parentheses after sayHi. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

It is a special value of course, in the sense that we can call it like sayHi().

But it’s still a value. So we can work with it like with other kinds of values.

We can copy a function to another variable:

function sayHi() { // (1) create

alert( "Hello" );

}

let func = sayHi; // (2) copy

func(); // Hello // (3) run the copy (it works)!

sayHi(); // Hello // this still works too (why wouldn't it)

Here’s what happens above in detail:

1. The Function Declaration (1) creates the function and puts it into the variable named sayHi.
2. Line (2) copies it into the variable func.

Please note again: there are no parentheses after sayHi. If there were, then func = sayHi() would write *the result of the call* sayHi() into func, not *the function* sayHi itself.

1. Now the function can be called as both sayHi() and func().

Note that we could also have used a Function Expression to declare sayHi, in the first line:

let sayHi = function() { ... };

let func = sayHi;

// ...

Everything would work the same. Even more obvious what’s going on, right?

**Why there’s a semicolon at the end?**

There might be a question, why does Function Expression have a semicolon ; at the end, and Function Declaration does not:

function sayHi() {

// ...

}

let sayHi = function() {

// ...

};

The answer is simple:

* There’s no need for ; at the end of code blocks and syntax structures that use them like if { ... }, for { }, function f { } etc.
* A Function Expression is used inside the statement: let sayHi = ...;, as a value. It’s not a code block. The semicolon ; is recommended at the end of statements, no matter what is the value. So the semicolon here is not related to the Function Expression itself in any way, it just terminates the statement.

**[Callback functions](https://javascript.info/function-expressions-arrows" \l "callback-functions)**

Let’s look at more examples of passing functions as values and using function expressions.

We’ll write a function ask(question, yes, no) with three parameters:

**question**

Text of the question

**yes**

Function to run if the answer is “Yes”

**no**

Function to run if the answer is “No”

The function should ask the question and, depending on the user’s answer, call yes() or no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "You agreed." );

}

function showCancel() {

alert( "You canceled the execution." );

}

// usage: functions showOk, showCancel are passed as arguments to ask

ask("Do you agree?", showOk, showCancel);

Before we explore how we can write it in a much shorter way, let’s note that in the browser (and on the server-side in some cases) such functions are quite popular. The major difference between a real-life implementation and the example above is that real-life functions use more complex ways to interact with the user than a simple confirm. In the browser, such a function usually draws a nice-looking question window. But that’s another story.

**The arguments of ask are called *callback functions* or just *callbacks*.**

The idea is that we pass a function and expect it to be “called back” later if necessary. In our case, showOk becomes the callback for the “yes” answer, and showCancel for the “no” answer.

We can use Function Expressions to write the same function much shorter:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Do you agree?",

function() { alert("You agreed."); },

function() { alert("You canceled the execution."); }

);

Here, functions are declared right inside the ask(...) call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of ask (because they are not assigned to variables), but that’s just what we want here.

Such code appears in our scripts very naturally, it’s in the spirit of JavaScript.

**A function is a value representing an “action”**

Regular values like strings or numbers represent the *data*.

A function can be perceived as an *action*.

We can pass it between variables and run when we want.

**[Function Expression vs Function Declaration](https://javascript.info/function-expressions-arrows" \l "function-expression-vs-function-declaration)**

Let’s formulate the key differences between Function Declarations and Expressions.

First, the syntax: how to see what is what in the code.

* *Function Declaration:* a function, declared as a separate statement, in the main code flow.
* // Function Declaration
* function sum(a, b) {
* return a + b;

}

* *Function Expression:* a function, created inside an expression or inside another syntax construct. Here, the function is created at the right side of the “assignment expression” =:
* // Function Expression
* let sum = function(a, b) {
* return a + b;

};

The more subtle difference is *when* a function is created by the JavaScript engine.

**A Function Expression is created when the execution reaches it and is usable from then on.**

Once the execution flow passes to the right side of the assignment let sum = function… – here we go, the function is created and can be used (assigned, called etc) from now on.

Function Declarations are different.

**A Function Declaration is usable in the whole script/code block.**

In other words, when JavaScript *prepares* to run the script or a code block, it first looks for Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all of the Function Declarations are processed, the execution goes on.

As a result, a function declared as a Function Declaration can be called earlier than it is defined.

For example, this works:

sayHi("John"); // Hello, John

function sayHi(name) {

alert( `Hello, ${name}` );

}

The Function Declaration sayHi is created when JavaScript is preparing to start the script and is visible everywhere in it.

…If it was a Function Expression, then it wouldn’t work:

sayHi("John"); // error!

let sayHi = function(name) { // (\*) no magic any more

alert( `Hello, ${name}` );

};

Function Expressions are created when the execution reaches them. That would happen only in the line (\*). Too late.

**When a Function Declaration is made within a code block, it is visible everywhere inside that block. But not outside of it.**

Sometimes that’s handy to declare a local function only needed in that block alone. But that feature may also cause problems.

For instance, let’s imagine that we need to declare a function welcome() depending on the age variable that we get during runtime. And then we plan to use it some time later.

The code below doesn’t work:

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

// conditionally declare a function

if (age < 18) {

function welcome() {

alert("Hello!");

}

} else {

function welcome() {

alert("Greetings!");

}

}

// ...use it later

welcome(); // Error: welcome is not defined

That’s because a Function Declaration is only visible inside the code block in which it resides.

Here’s another example:

let age = 16; // take 16 as an example

if (age < 18) {

welcome(); // \ (runs)

// |

function welcome() { // |

alert("Hello!"); // | Function Declaration is available

} // | everywhere in the block where it's declared

// |

welcome(); // / (runs)

} else {

function welcome() { // for age = 16, this "welcome" is never created

alert("Greetings!");

}

}

// Here we're out of curly braces,

// so we can not see Function Declarations made inside of them.

welcome(); // Error: welcome is not defined

What can we do to make welcome visible outside of if?

The correct approach would be to use a Function Expression and assign welcome to the variable that is declared outside of ifand has the proper visibility.

Now it works as intended:

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

let welcome;

if (age < 18) {

welcome = function() {

alert("Hello!");

};

} else {

welcome = function() {

alert("Greetings!");

};

}

welcome(); // ok now

Or we could simplify it even further using a question mark operator ?:

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

let welcome = (age < 18) ?

function() { alert("Hello!"); } :

function() { alert("Greetings!"); };

welcome(); // ok now

**When to choose Function Declaration versus Function Expression?**

As a rule of thumb, when we need to declare a function, the first to consider is Function Declaration syntax, the one we used before. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

It’s also a little bit easier to look up function f(…) {…} in the code than let f = function(…) {…}. Function Declarations are more “eye-catching”.

…But if a Function Declaration does not suit us for some reason (we’ve seen an example above), then Function Expression should be used.

**[Arrow functions](https://javascript.info/function-expressions-arrows" \l "arrow-functions)**

There’s one more 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 has arguments arg1..argN, evaluates the expression on the right side with their use and returns its result.

In other words, it’s roughly the same as:

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

return expression;

}

…But much more concise.

Let’s see an example:

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

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

let sum = function(a, b) {

return a + b;

};

\*/

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

If we have only one argument, then parentheses can be omitted, making that even shorter:

// same as

// let double = function(n) { return n \* 2 }

let double = n => n \* 2;

alert( double(3) ); // 6

If there are no arguments, parentheses should 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, here’s the rewritten example with welcome():

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**

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, use return to get results

};

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. We’ll return to them later in the chapter [Arrow functions revisited](https://javascript.info/arrow-functions).

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

[**Summary**](https://javascript.info/function-expressions-arrows#summary)

* 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 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 return to return something.

[**Tasks**](https://javascript.info/function-expressions-arrows#tasks)

**[Rewrite with arrow functions](https://javascript.info/function-expressions-arrows" \l "rewrite-with-arrow-functions)**

Replace Function Expressions with arrow functions in the code:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Do you agree?",

function() { alert("You agreed."); },

function() { alert("You canceled the execution."); }

);

# JavaScript specials

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

## [Code structure](https://javascript.info/javascript-specials" \l "code-structure)

Statements are delimited with a semicolon:

alert('Hello'); alert('World');

Usually, a line-break is also treated as a delimiter, so that would also work:

alert('Hello')

alert('World')

That’s called “automatic semicolon insertion”. Sometimes it doesn’t work, for instance:

alert("There will be an error after this message")

[1, 2].forEach(alert)

Most codestyle guides agree that we should put a semicolon after each statement.

Semicolons are not required after code blocks {...} and syntax constructs with them like loops:

function f() {

// no semicolon needed after function declaration

}

for(;;) {

// no semicolon needed after the loop

}

…But even if we can put an “extra” semicolon somewhere, that’s not an error. It will be ignored.

More in: [Code structure](https://javascript.info/structure).

## [Strict mode](https://javascript.info/javascript-specials" \l "strict-mode)

To fully enable all features of modern JavaScript, we should start scripts with "use strict".

'use strict';

...

The directive must be at the top of a script or at the beginning of a function.

Without "use strict", everything still works, but some features behave in the old-fashion, “compatible” way. We’d generally prefer the modern behavior.

Some modern features of the language (like classes that we’ll study in the future) enable strict mode implicitly.

More in: [The modern mode, "use strict"](https://javascript.info/strict-mode).

## [Variables](https://javascript.info/javascript-specials" \l "variables)

Can be declared using:

* let
* const (constant, can’t be changed)
* var (old-style, will see later)

A variable name can include:

* Letters and digits, but the first character may not be a digit.
* Characters $ and \_ are normal, on par with letters.
* Non-Latin alphabets and hieroglyphs are also allowed, but commonly not used.

Variables are dynamically typed. They can store any value:

let x = 5;

x = "John";

There are 7 data types:

* number for both floating-point and integer numbers,
* string for strings,
* boolean for logical values: true/false,
* null – a type with a single value null, meaning “empty” or “does not exist”,
* undefined – a type with a single value undefined, meaning “not assigned”,
* object and symbol – for complex data structures and unique identifiers, we haven’t learnt them yet.

The typeof operator returns the type for a value, with two exceptions:

typeof null == "object" // error in the language

typeof function(){} == "function" // functions are treated specially

More in: [Variables](https://javascript.info/variables) and [Data types](https://javascript.info/types).

## [Interaction](https://javascript.info/javascript-specials" \l "interaction)

We’re using a browser as a working environment, so basic UI functions will be:

[**prompt(question[, default])**](https://developer.mozilla.org/en-US/docs/Web/API/Window/prompt)

Ask a question, and return either what the visitor entered or null if he pressed “cancel”.

[**confirm(question)**](https://developer.mozilla.org/en-US/docs/Web/API/Window/confirm)

Ask a question and suggest to choose between Ok and Cancel. The choice is returned as true/false.

[**alert(message)**](https://developer.mozilla.org/en-US/docs/Web/API/Window/alert)

Output a message.

All these functions are modal, they pause the code execution and prevent the visitor from interacting with the page until he answers.

For instance:

let userName = prompt("Your name?", "Alice");

let isTeaWanted = confirm("Do you want some tea?");

alert( "Visitor: " + userName ); // Alice

alert( "Tea wanted: " + isTeaWanted ); // true

More in: [Interaction: alert, prompt, confirm](https://javascript.info/alert-prompt-confirm).

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

JavaScript supports the following operators:

**Arithmetical**

Regular: \* + - /, also % for the remainder and \*\* for power of a number.

The binary plus + concatenates strings. And if any of the operands is a string, the other one is converted to string too:

alert( '1' + 2 ); // '12', string

alert( 1 + '2' ); // '12', string

**Assignments**

There is a simple assignment: a = b and combined ones like a \*= 2.

**Bitwise**

Bitwise operators work with integers on bit-level: see the [docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) when they are needed.

**Ternary**

The only operator with three parameters: cond ? resultA : resultB. If cond is truthy, returns resultA, otherwise resultB.

**Logical operators**

Logical AND && and OR || perform short-circuit evaluation and then return the value where it stopped.

**Comparisons**

Equality check == for values of different types converts them to a number (except null and undefined that equal each other and nothing else), so these are equal:

alert( 0 == false ); // true

alert( 0 == '' ); // true

Other comparisons convert to a number as well.

The strict equality operator === doesn’t do the conversion: different types always mean different values for it, so:

Values null and undefined are special: they equal == each other and don’t equal anything else.

Greater/less comparisons compare strings character-by-character, other types are converted to a number.

**Logical operators**

There are few others, like a comma operator.

More in: [Operators](https://javascript.info/operators), [Comparisons](https://javascript.info/comparison), [Logical operators](https://javascript.info/logical-operators).

## [Loops](https://javascript.info/javascript-specials" \l "loops)

* We covered 3 types of loops:
* // 1
* while (condition) {
* ...
* }
* // 2
* do {
* ...
* } while (condition);
* // 3
* for(let i = 0; i < 10; i++) {
* ...

}

* The variable declared in for(let...) loop is visible only inside the loop. But we can also omit let and reuse an existing variable.
* Directives break/continue allow to exit the whole loop/current iteration. Use labels to break nested loops.

Details in: [Loops: while and for](https://javascript.info/while-for).

Later we’ll study more types of loops to deal with objects.

## [The “switch” construct](https://javascript.info/javascript-specials" \l "the-switch-construct)

The “switch” construct can replace multiple if checks. It uses === (strict equality) for comparisons.

For instance:

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

switch (age) {

case 18:

alert("Won't work"); // the result of prompt is a string, not a number

case "18":

alert("This works!");

break;

default:

alert("Any value not equal to one above");

}

Details in: [The "switch" statement](https://javascript.info/switch).

# Coding style

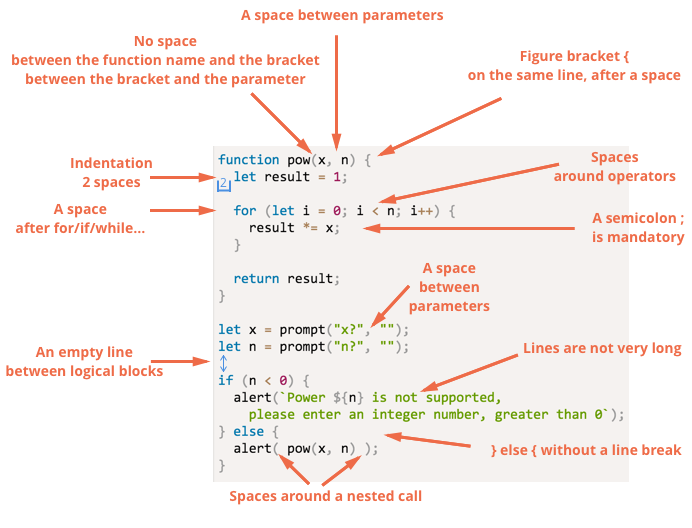
Our code must be as clean and easy to read as possible.

That is actually an art of programming – to take a complex task and code it in a way that is both correct and human-readable.

One thing to help is the good code style.

## [Syntax](https://javascript.info/coding-style" \l "syntax)

A cheatsheet with the rules (more details below):



Now let’s discuss the rules and reasons for them in detail.

Nothing is “carved in stone” here. Everything is optional and can be changed: these are coding rules, not religious dogmas.

### [Curly braces](https://javascript.info/coding-style" \l "curly-braces)

In most JavaScript projects curly braces are written on the same line as the corresponding keyword, not on the new line, a so-called “Egyptian” style. There’s also a space before an opening bracket.

Like this:

if (condition) {

// do this

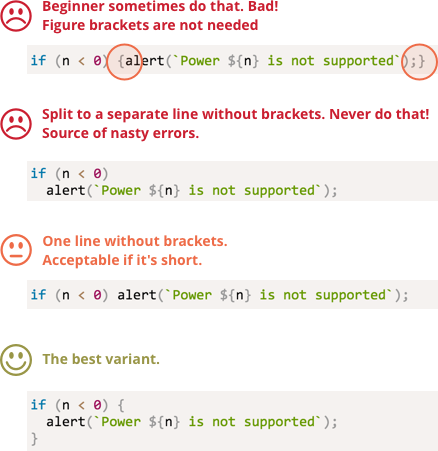
// ...and that

// ...and that

}

A single-line construct is an important edge case. Should we use brackets at all? If yes, then where?

Here are the annotated variants, so you can judge about their readability on your own:



As a summary:

* For a really short code, one line is acceptable: like if (cond) return null.
* But a separate line for each statement in brackets is usually better.

### [Line length](https://javascript.info/coding-style" \l "line-length)

The maximal line length should be limited. No one likes to eye-follow a long horizontal line. It’s better to split it.

The maximal line length is agreed on the team-level. It’s usually 80 or 120 characters.

### [Indents](https://javascript.info/coding-style" \l "indents)

There are two types of indents:

* **A horizontal indent: 2(4) spaces.**

A horizontal indentation is made using either 2 or 4 spaces or the “Tab” symbol. Which one to choose is an old holy war. Spaces are more common nowadays.

One advantage of spaces over tabs is that spaces allow more flexible configurations of indents than the “Tab” symbol.

For instance, we can align the arguments with the opening bracket, like this:

show(parameters,

aligned, // 5 spaces padding at the left

one,

after,

another

) {

// ...

}

* **A vertical indent: empty lines for splitting code into logical blocks.**

Even a single function can often be divided in logical blocks. In the example below, the initialization of variables, the main loop and returning the result are split vertically:

function pow(x, n) {

let result = 1;

// <--

for (let i = 0; i < n; i++) {

result \*= x;

}

// <--

return result;

}

Insert an extra newline where it helps to make the code more readable. There should not be more than nine lines of code without a vertical indentation.

### [A semicolon](https://javascript.info/coding-style" \l "a-semicolon)

A semicolon should be present after each statement. Even if it could possibly be skipped.

There are languages where a semicolon is truly optional. It’s rarely used there. But in JavaScript there are few cases when a line break is sometimes not interpreted as a semicolon. That leaves a place for programming errors.

As you become more mature as a programmer, you may choose a no-semicolon style, like [StandardJS](https://standardjs.com/), but that’s only when you know JavaScript well and understand possible pitfalls.

### [Nesting levels](https://javascript.info/coding-style" \l "nesting-levels)

There should not be too many nesting levels.

Sometimes it’s a good idea to use the [“continue”](https://javascript.info/while-for#continue) directive in the loop to evade extra nesting in if(..) { ... }:

Instead of:

for (let i = 0; i < 10; i++) {

if (cond) {

... // <- one more nesting level

}

}

We can write:

for (let i = 0; i < 10; i++) {

if (!cond) continue;

... // <- no extra nesting level

}

A similar thing can be done with if/else and return.

For example, two constructs below are identical.

The first one:

function pow(x, n) {

if (n < 0) {

alert("Negative 'n' not supported");

} else {

let result = 1;

for (let i = 0; i < n; i++) {

result \*= x;

}

return result;

}

}

And this:

function pow(x, n) {

if (n < 0) {

alert("Negative 'n' not supported");

return;

}

let result = 1;

for (let i = 0; i < n; i++) {

result \*= x;

}

return result;

}

…But the second one is more readable, because the “edge case” of n < 0 is handled early on, and then we have the “main” code flow, without an additional nesting.

## [Functions below the code](https://javascript.info/coding-style" \l "functions-below-the-code)

If you are writing several “helper” functions and the code to use them, then there are three ways to place them.

1. Functions above the code that uses them:
2. // function declarations
3. function createElement() {
4. ...
5. }
6. function setHandler(elem) {
7. ...
8. }
9. function walkAround() {
10. ...
11. }
12. // the code which uses them
13. let elem = createElement();
14. setHandler(elem);

walkAround();

1. Code first, then functions
2. // the code which uses the functions
3. let elem = createElement();
4. setHandler(elem);
5. walkAround();
6. // --- helper functions ---
7. function createElement() {
8. ...
9. }
10. function setHandler(elem) {
11. ...
12. }
13. function walkAround() {
14. ...

}

1. Mixed: a function is described where it’s first used.

Most of time, the second variant is preferred.

That’s because when reading a code, we first want to know “what it does”. If the code goes first, then it provides that information. And then maybe we won’t need to read functions at all, especially if their names are adequate to what they’re doing.

## [Style guides](https://javascript.info/coding-style" \l "style-guides)

A style guide contains general rules about “how to write”: which quotes to use, how many spaces to indent, where to put line breaks, etc. A lot of minor things.

In total, when all members of a team use the same style guide, the code looks uniform. No matter who of the team wrote it, it’s still the same style.

Surely, a team may think out a style guide themselves. But as of now, there’s no need to. There are many tried, worked-out style guides, which are easy to adopt.

For instance:

* [Google JavaScript Style Guide](https://google.github.io/styleguide/javascriptguide.xml)
* [Airbnb JavaScript Style Guide](https://github.com/airbnb/javascript)
* [Idiomatic.JS](https://github.com/rwaldron/idiomatic.js)
* [StandardJS](https://standardjs.com/)
* (there are more)

If you’re a novice developer, then you could start with the cheatsheet above in the chapter, and later browse the style guides to pick up the common principles and maybe choose one.

## [Automated linters](https://javascript.info/coding-style" \l "automated-linters)

There are tools that can check the code style automatically. They are called “linters”.

The great thing about them is that style-checking also finds some bugs, like a typo in a variable or function name.

So it’s recommended to install one, even if you don’t want to stick to a “code style”. They help to find typos – and that’s already good enough.

Most well-known tools are:

* [JSLint](http://www.jslint.com/) – one of the first linters.
* [JSHint](http://www.jshint.com/) – more settings than JSLint.
* [ESLint](http://eslint.org/) – probably the newest one.

All of them can do the job. The author uses [ESLint](http://eslint.org/).

Most linters are integrated with editors: just enable the plugin in the editor and configure the style.

For instance, for ESLint you should do the following:

1. Install [Node.JS](https://nodejs.org/).
2. Install ESLint with the command npm install -g eslint (npm is a JavaScript package installer).
3. Create a config file named .eslintrc in the root of your JavaScript project (in the folder that contains all your files).

Here’s an example of .eslintrc:

{

"extends": "eslint:recommended",

"env": {

"browser": true,

"node": true,

"es6": true

},

"rules": {

"no-console": 0,

},

"indent": 2

}

Here the directive "extends" denotes that we base on the “eslint:recommended” set of settings, and then we specify our own.

Then install/enable the plugin for your editor that integrates with ESLint. The majority of editors have it.

It is possible to download style rule sets from the web and extend them instead. See <http://eslint.org/docs/user-guide/getting-started> for more details about installation.

Using a linter has a great side-effect: linters catch typos. For instance, when an undefined variable is accessed, a linter detects it and (if integrated with an editor) highlights it. In most cases that’s a mistype. So we can fix it right ahead.

For that reason even if you’re not concerned about styles, using a linter is highly recommended.

Also certain IDEs support built-in linting, that also may be good, but not so tunable as ESLint.

## [Summary](https://javascript.info/coding-style#summary)

All syntax rules from this chapter and the style guides aim to increase readability, so all of them are debatable.

When we think about “how to write better?”, the sole criterion is “what makes the code more readable and easier to understand? what helps to avoid errors?” That’s the main thing to keep in mind when choosing the style or discussing which one is better.

Read style guides to see the latest ideas about that and follow those that you find the best.