**Contents**

[Introduction to JavaScript 8](#_Toc137725581)

[What is JavaScript? 8](#_Toc137725582)

[Why is it called JavaScript? 8](#_Toc137725583)

[How do engines work? 8](#_Toc137725584)

[What can in-browser JavaScript do? 9](#_Toc137725585)

[What CAN’T in-browser JavaScript do? 9](#_Toc137725586)

[What makes JavaScript unique? 10](#_Toc137725587)

[Hello, world! 11](#_Toc137725588)

[The “script” tag 11](#_Toc137725589)

[Modern markup 11](#_Toc137725590)

[External scripts 12](#_Toc137725591)

[Summary 13](#_Toc137725592)

[Tasks 13](#_Toc137725593)

[Show an alert 13](#_Toc137725594)

[Show an alert with an external script 13](#_Toc137725595)

[Code structure 15](#_Toc137725596)

[Statements 15](#_Toc137725597)

[Semicolons 15](#_Toc137725598)

[Comments 16](#_Toc137725599)

[The modern mode, "use strict" 19](#_Toc137725600)

[“use strict” 19](#_Toc137725601)

[Browser console 20](#_Toc137725602)

[Should we “use strict”? 20](#_Toc137725603)

[Variables 21](#_Toc137725604)

[A variable 21](#_Toc137725605)

[A real-life analogy 22](#_Toc137725606)

[Variable naming 23](#_Toc137725607)

[Constants 25](#_Toc137725608)

[Uppercase constants 25](#_Toc137725609)

[Name things right 26](#_Toc137725610)

[Summary 27](#_Toc137725611)

[Tasks 27](#_Toc137725612)

[Working with variables 27](#_Toc137725613)

[Giving the right name 28](#_Toc137725614)

[Uppercase const? 28](#_Toc137725615)

[Data types 30](#_Toc137725616)

[Number 30](#_Toc137725617)

[BigInt 31](#_Toc137725618)

[String 31](#_Toc137725619)

[Boolean (logical type) 32](#_Toc137725620)

[The “null” value 33](#_Toc137725621)

[The “undefined” value 33](#_Toc137725622)

[Objects and Symbols 34](#_Toc137725623)

[The typeof operator 34](#_Toc137725624)

[Summary 35](#_Toc137725625)

[Tasks 35](#_Toc137725626)

[String quotes 35](#_Toc137725627)

[Comparisons 37](#_Toc137725628)

[Boolean is the result 37](#_Toc137725629)

[String comparison 37](#_Toc137725630)

[Comparison of different types 38](#_Toc137725631)

[Strict equality 39](#_Toc137725632)

[Comparison with null and undefined 39](#_Toc137725633)

[Strange result: null vs 0 40](#_Toc137725634)

[An incomparable undefined 40](#_Toc137725635)

[Avoid problems 40](#_Toc137725636)

[Summary 41](#_Toc137725637)

[Tasks 41](#_Toc137725638)

[Comparisons 41](#_Toc137725639)

[Conditional branching: if, '?' 43](#_Toc137725640)

[The “if” statement 43](#_Toc137725641)

[Boolean conversion 43](#_Toc137725642)

[The “else” clause 44](#_Toc137725643)

[Several conditions: “else if” 44](#_Toc137725644)

[Conditional operator ‘?’ 45](#_Toc137725645)

[Multiple ‘?’ 46](#_Toc137725646)

[Non-traditional use of ‘?’ 46](#_Toc137725647)

[Tasks 47](#_Toc137725648)

[if (a string with zero) 47](#_Toc137725649)

[The name of JavaScript 48](#_Toc137725650)

[Show the sign 49](#_Toc137725651)

[Rewrite 'if' into '?' 49](#_Toc137725652)

[Rewrite 'if..else' into '?' 50](#_Toc137725653)

[Logical operators 51](#_Toc137725654)

[|| (OR) 51](#_Toc137725655)

[OR "||" finds the first truthy value 52](#_Toc137725656)

[&& (AND) 53](#_Toc137725657)

[AND “&&” finds the first falsy value 54](#_Toc137725658)

[! (NOT) 55](#_Toc137725659)

[Tasks 56](#_Toc137725660)

[What's the result of OR? 56](#_Toc137725661)

[What's the result of OR'ed alerts? 56](#_Toc137725662)

[What is the result of AND? 56](#_Toc137725663)

[What is the result of AND'ed alerts? 57](#_Toc137725664)

[The result of OR AND OR 57](#_Toc137725665)

[Check the range between 57](#_Toc137725666)

[Check the range outside 58](#_Toc137725667)

[A question about "if" 58](#_Toc137725668)

[Check the login 59](#_Toc137725669)

[Loops: while and for 61](#_Toc137725670)

[The “while” loop 61](#_Toc137725671)

[The “do…while” loop 62](#_Toc137725672)

[The “for” loop 62](#_Toc137725673)

[Skipping parts 64](#_Toc137725674)

[Breaking the loop 65](#_Toc137725675)

[Continue to the next iteration 65](#_Toc137725676)

[Labels for break/continue 66](#_Toc137725677)

[Summary 68](#_Toc137725678)

[Tasks 68](#_Toc137725679)

[Last loop value 68](#_Toc137725680)

[Which values does the while loop show? 69](#_Toc137725681)

[Which values get shown by the "for" loop? 70](#_Toc137725682)

[Output even numbers in the loop 71](#_Toc137725683)

[Replace "for" with "while" 71](#_Toc137725684)

[Repeat until the input is correct 72](#_Toc137725685)

[Output prime numbers 72](#_Toc137725686)

[Functions 74](#_Toc137725687)

[Function Declaration 74](#_Toc137725688)

[Local variables 75](#_Toc137725689)

[Outer variables 75](#_Toc137725690)

[Parameters 76](#_Toc137725691)

[Default values 77](#_Toc137725692)

[Alternative default parameters 79](#_Toc137725693)

[Returning a value 80](#_Toc137725694)

[Naming a function 82](#_Toc137725695)

[Functions == Comments 83](#_Toc137725696)

[Summary 84](#_Toc137725697)

[Tasks 85](#_Toc137725698)

[Is "else" required? 85](#_Toc137725699)

[Rewrite the function using '?' or '||' 85](#_Toc137725700)

[Function min(a, b) 86](#_Toc137725701)

[Function pow(x,n) 87](#_Toc137725702)

[JavaScript – this 88](#_Toc137725703)

[Global Scope 88](#_Toc137725704)

[Object Method 88](#_Toc137725705)

[Class basic syntax 90](#_Toc137725706)

[The “class” syntax 90](#_Toc137725707)

[What is a class? 91](#_Toc137725708)

[Not just a syntactic sugar 92](#_Toc137725709)

[Class Expression 93](#_Toc137725710)

[Getters/setters 94](#_Toc137725711)

[Computed names […] 95](#_Toc137725712)

[Class fields 96](#_Toc137725713)

[Making bound methods with class fields 97](#_Toc137725714)

[Summary 98](#_Toc137725715)

[Tasks 98](#_Toc137725716)

[Rewrite to class 98](#_Toc137725717)

[Class inheritance 100](#_Toc137725718)

[The “extends” keyword 100](#_Toc137725719)

[Overriding a method 102](#_Toc137725720)

[Overriding constructor 104](#_Toc137725721)

[Overriding class fields: a tricky note 106](#_Toc137725722)

[Super: internals, [[HomeObject]] 108](#_Toc137725723)

[[[HomeObject]] 111](#_Toc137725724)

[Methods are not “free” 112](#_Toc137725725)

[Methods, not function properties 113](#_Toc137725726)

[Summary 114](#_Toc137725727)

[Tasks 114](#_Toc137725728)

[Error creating an instance 114](#_Toc137725729)

[Extended clock 115](#_Toc137725730)

[JavaScript – ES6 118](#_Toc137725731)

[JavaScript let 118](#_Toc137725732)

[JavaScript const 118](#_Toc137725733)

[Arrow Functions 119](#_Toc137725734)

[The Spread (...) Operator 120](#_Toc137725735)

[The For/Of Loop 121](#_Toc137725736)

[Looping over an Array 121](#_Toc137725737)

[Looping over a String 122](#_Toc137725738)

[JavaScript Maps 122](#_Toc137725739)

[JavaScript Sets 123](#_Toc137725740)

[JavaScript - Modules 124](#_Toc137725741)

[What is a module? 124](#_Toc137725742)

[Core module features 125](#_Toc137725743)

[Always “use strict” 125](#_Toc137725744)

[Module-level scope 126](#_Toc137725745)

[A module code is evaluated only the first time when imported 127](#_Toc137725746)

[import.meta 129](#_Toc137725747)

[In a module, “this” is undefined 129](#_Toc137725748)

[Browser-specific features 130](#_Toc137725749)

[Module scripts are deferred 130](#_Toc137725750)

[Async works on inline scripts 131](#_Toc137725751)

[External scripts 131](#_Toc137725752)

[No “bare” modules allowed 132](#_Toc137725753)

[Compatibility, “nomodule” 132](#_Toc137725754)

[Build tools 132](#_Toc137725755)

[Summary 133](#_Toc137725756)

[JavaScript - DOM 134](#_Toc137725757)

[An example of the DOM 134](#_Toc137725758)

[Autocorrection 136](#_Toc137725759)

[Other node types 137](#_Toc137725760)

[See it for yourself 139](#_Toc137725761)

[Interaction with console 140](#_Toc137725762)

[Summary 142](#_Toc137725763)

[Walking the DOM 143](#_Toc137725764)

[On top: documentElement and body 143](#_Toc137725765)

[Children: childNodes, firstChild, lastChild 144](#_Toc137725766)

[DOM collections 145](#_Toc137725767)

[Siblings and the parent 146](#_Toc137725768)

[Element-only navigation 147](#_Toc137725769)

[More links: tables 148](#_Toc137725770)

[Summary 149](#_Toc137725771)

[Tasks 149](#_Toc137725772)

[DOM children 149](#_Toc137725773)

[The sibling question 150](#_Toc137725774)

[Select all diagonal cells 151](#_Toc137725775)

[Searching: getElement\*, querySelector\* 154](#_Toc137725776)

[document.getElementById or just id 154](#_Toc137725777)

[querySelectorAll 155](#_Toc137725778)

[querySelector 156](#_Toc137725779)

[matches 156](#_Toc137725780)

[closest 156](#_Toc137725781)

[getElementsBy\* 157](#_Toc137725782)

[Live collections 158](#_Toc137725783)

[Summary 159](#_Toc137725784)

[Tasks 160](#_Toc137725785)

[Search for elements 160](#_Toc137725786)

[JavaScript – Events 163](#_Toc137725787)

[Event handlers 163](#_Toc137725788)

[HTML-attribute 163](#_Toc137725789)

[DOM property 164](#_Toc137725790)

[Accessing the element: this 165](#_Toc137725791)

[Possible mistakes 165](#_Toc137725792)

[addEventListener 166](#_Toc137725793)

[Event object 168](#_Toc137725794)

[Object handlers: handleEvent 169](#_Toc137725795)

[Summary 170](#_Toc137725796)

[Tasks 171](#_Toc137725797)

[JavaScript – Event Propagation – Bubbling and Capturing 172](#_Toc137725798)

[Bubbling 172](#_Toc137725799)

[event.target 173](#_Toc137725800)

[Stopping bubbling 175](#_Toc137725801)

[Capturing 176](#_Toc137725802)

[Summary 178](#_Toc137725803)

[JavaScript – Storage – Cookies, document.cookie 180](#_Toc137725804)

[JavaScript – Storage – LocalStorage, sessionStorage 181](#_Toc137725805)

[localStorage demo 181](#_Toc137725806)

[Object-like access 182](#_Toc137725807)

[Looping over keys 182](#_Toc137725808)

[Strings only 183](#_Toc137725809)

[sessionStorage 183](#_Toc137725810)

[Storage event 184](#_Toc137725811)

[Summary 184](#_Toc137725812)

[Tasks 185](#_Toc137725813)

[Autosave a form field 185](#_Toc137725814)

[JavaScript – Promises and Callbacks 187](#_Toc137725815)

[Introduction to Callbacks 187](#_Toc137725816)

[Callback in callback 188](#_Toc137725817)

[Handling errors 189](#_Toc137725818)

[Pyramid of Doom 190](#_Toc137725819)

[Promise 192](#_Toc137725820)

[Consumers: then, catch 195](#_Toc137725821)

[Cleanup: finally 196](#_Toc137725822)

[Example: loadScript 198](#_Toc137725823)

[Tasks 199](#_Toc137725824)

[How Promises Work in JavaScript – A Comprehensive Beginner's Guide 202](#_Toc137725825)

[Why Should You Care about Promises? 202](#_Toc137725826)

[What is a Promise? 204](#_Toc137725827)

[How to Create a Promise in JavaScript 205](#_Toc137725828)

[How to Attach a Callback to a Promise 205](#_Toc137725829)

[How to Handle Errors in a Promise 208](#_Toc137725830)

[How to Handle Many Promises at Once 209](#_Toc137725831)

[The Promise.all() method 209](#_Toc137725832)

[The Promise.race() method 211](#_Toc137725833)

[The Promise.any() method 212](#_Toc137725834)

[The Promise.allSettled() method 214](#_Toc137725835)

[What is the Async/Await Syntax? 216](#_Toc137725836)

[How to Create an Async Function in JavaScript 216](#_Toc137725837)

[How to Use the Await Keyword 217](#_Toc137725838)

[How to Handle Errors in Async/Await 218](#_Toc137725839)

[What is a Job Queue? 219](#_Toc137725840)

[JavaScript – AJAX 221](#_Toc137725841)

[What is Ajax? 221](#_Toc137725842)

[Understanding How Ajax Works 221](#_Toc137725843)

[Performing an Ajax GET Request 222](#_Toc137725844)

[Performing an Ajax POST Request 225](#_Toc137725845)

[Fetch 227](#_Toc137725846)

[JavaScript – JSON 229](#_Toc137725847)

[JSON Data - A Name and a Value 229](#_Toc137725848)

[JSON Objects 229](#_Toc137725849)

[JSON Arrays 229](#_Toc137725850)

[Converting a JSON Text to a JavaScript Object 230](#_Toc137725851)

[HTTP – CORS 231](#_Toc137725852)

[What is CORS (cross-origin resource sharing)? 231](#_Toc137725853)

[Same-origin policy 231](#_Toc137725854)

[header 232](#_Toc137725855)

[body 232](#_Toc137725856)

[How does CORS work? 232](#_Toc137725857)

[See CORS in action 232](#_Toc137725858)

# Introduction to JavaScript

<https://javascript.info/intro>

## What is JavaScript?

JavaScript was initially created to “make web pages alive”.

The programs in this language are called scripts. They can be written right in a web page’s HTML and run automatically as the page loads.

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

In this aspect, JavaScript is very different from another language called Java.

### Why is it called JavaScript?

When JavaScript was created, it initially had another name: “LiveScript”. But Java 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.

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

The browser has an embedded engine sometimes called a “JavaScript virtual machine”.

Different engines have different “codenames”. For example:

* [V8](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) – in Chrome, Opera and Edge.
* [SpiderMonkey](https://en.wikipedia.org/wiki/SpiderMonkey) – in Firefox.
* …There are other codenames like “Chakra” for IE, “JavaScriptCore”, “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, Opera and Edge.

### How do 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 machine code.
3. And then the machine code runs, pretty fast.

The engine applies optimizations at each step of the process. It even watches the compiled script as it runs, analyzes the data that flows through it, and further optimizes the machine code based on that knowledge.

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

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

JavaScript’s capabilities greatly depend on the environment it’s running in. 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 to protect the user’s safety. The aim is to prevent an evil webpage from accessing private information or harming the user’s data.

Examples of such restrictions include:

* 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 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.

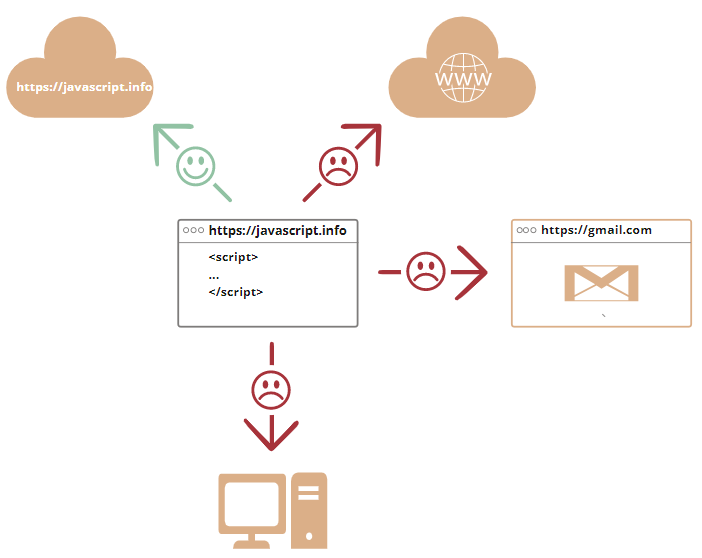
There are ways to interact with the camera/microphone and other devices, but they require a user’s explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the [NSA](https://en.wikipedia.org/wiki/National_Security_Agency).

* 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 page 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 agree for data exchange and must contain special JavaScript code that handles it. We’ll cover that in the tutorial.

This limitation is, again, for the 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, for example, 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 a safety limitation.



Such limitations do not exist if JavaScript is used outside of the browser, for example on a server. Modern browsers also allow plugins/extensions which may ask for 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 are done simply.
* Supported by all major browsers and enabled by default.

JavaScript is the only browser technology that combines these three things.

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

That said, JavaScript can be used to create servers, mobile applications, etc.

# Hello, world!

This part of the tutorial is about core JavaScript, the language itself.

But we need a working environment to run our scripts and, since this book is online, the browser is a good choice. We’ll keep the amount of browser-specific commands (like alert) to a minimum so that you don’t spend time on them if you plan to concentrate on another environment (like Node.js).

So first, let’s see how we attach a script to a webpage. For server-side environments (like Node.js), you can execute the script with a command like "node my.js".

## The “script” tag

JavaScript programs can be inserted almost anywhere into an HTML document using the <script> tag.

For instance:

<!DOCTYPE HTML>

<html>

<body>

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

<script>

alert( 'Hello, world!' );

</script>

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

</body>

</html>

You can run the example by clicking the “Play” button in the right-top corner of the box above.

The <script> tag contains JavaScript code which is automatically executed when the browser processes the tag.

## Modern markup

The <script> tag has a few attributes that are rarely used nowadays but can still be found in old code:

**The type attribute: <script type=…>**

The old HTML standard, HTML4, required a script to have a type. Usually it was type="text/javascript". It’s not required anymore. Also, the modern HTML standard totally changed the meaning of this attribute. Now, it can be used for JavaScript modules. But that’s an advanced topic, we’ll talk about modules in another part of the tutorial.

The **language** attribute: **<script language=…>**

This attribute was meant to show the language of the script. This attribute no longer makes sense because JavaScript is the default language. There is no need to use it.

**Comments before and after scripts.**

In really ancient books and guides, you may find comments inside <script> tags, like this:

<script type="text/javascript"><!--

...

//--></script>

This trick isn’t used in modern JavaScript. These comments hide JavaScript code from old browsers that didn’t know how to process the <script> tag. Since browsers released in the last 15 years don’t have this issue, this kind of comment can help you identify really old code.

## External scripts

If we have a lot of JavaScript code, we can put it into a separate file.

Script files are attached to HTML with the src attribute:

<script src="/path/to/script.js"></script>

Here, /path/to/script.js is an absolute path to the script from the site root. One can also provide a relative path from the current page. For instance, src="script.js", just like src="./script.js", would mean a file "script.js" in the current folder.

We can give a full URL as well. For instance:

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>

To attach several scripts, use multiple tags:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

…

**Please note:**

As a rule, only the simplest scripts are put into HTML. More complex ones reside in separate files.

The benefit of a separate file is that the browser will download it and store it in its cache.

Other pages that reference the same script will take it from the cache instead of downloading it, so the file is actually downloaded only once.

That reduces traffic and makes pages faster.

**If src is set, the script content is ignored.**

A single <script> tag can’t have both the src attribute and code inside.

This won’t work:

<script src="file.js">

alert(1); // the content is ignored, because src is set

</script>

We must choose either an external <script src="…"> or a regular <script> with code.

The example above can be split into two scripts to work:

<script src="file.js"></script>

<script>

alert(1);

</script>

## Summary

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

## Tasks

### Show an alert

importance: 5

Create a page that shows a message “I’m JavaScript!”.

Do it in a sandbox, or on your hard drive, doesn’t matter, just ensure that it works.

**Solution**

<!DOCTYPE html>

<html>

<body>

<script>

alert( "I'm JavaScript!" );

</script>

</body>

</html>

### Show an alert with an external script

importance: 5

Take the solution of the previous task [Show an alert](https://javascript.info/task/hello-alert). Modify it by extracting the script content into an external file alert.js, residing in the same folder.

Open the page, ensure that the alert works.

**Solution**

<!DOCTYPE html>

<html>

<body>

<script src="alert.js"></script>

</body>

</html>

For the file alert.js in the same folder:

alert("I'm JavaScript!");

# Code structure

The first thing we’ll study is the building blocks of code.

## Statements

Statements are syntax constructs and commands that perform actions.

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

We can have as many statements in our code as we want. Statements can be separated with a semicolon.

For example, here we split “Hello World” into two alerts:

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

Usually, statements are written on separate lines to make the code more readable:

alert('Hello');

alert('World');

## Semicolons

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. This is 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”, so a semicolon there would be incorrect. 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:

alert("Hello");

[1, 2].forEach(alert);

No need to think about the meaning of the brackets [] and forEach yet. We’ll study them later. For now, just remember the result of running the code: it shows Hello, then 1, then 2.

Now let’s remove the semicolon after the alert:

alert("Hello")

[1, 2].forEach(alert);

The difference compared to the code above is only one character: the semicolon at the end of the first line is gone.

If we run this code, only the first Hello shows (and there’s an error, you may need to open the console to see it). There are no numbers any more.

That’s because JavaScript does not assume a semicolon before square brackets [...]. So, the code in the last example is treated as a single statement.

Here’s how the engine sees it:

alert("Hello")[1, 2].forEach(alert);

Looks weird, right? Such merging in this case is just wrong. We need to put a semicolon after alert for the code to work correctly.

This can happen in other situations also.

We recommend putting 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

As time goes on, programs become more and more complex. It becomes necessary to add comments which describe what the code does and why.

Comments can be put into any place of a script. They don’t affect its 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 can be 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 pressing the 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 and Option instead of Shift.

**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 code before publishing to a production server. They remove comments, so they don’t appear in the working scripts. Therefore, comments do not have negative effects on production at all.

# The modern mode, "use strict"

For a long time, JavaScript evolved without compatibility issues. New features were added to the language while old functionality didn’t change.

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

This was the case 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 such modifications are off by default. You need to explicitly enable them with a special directive: "use strict".

## “use strict”

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

For example:

"use strict";

// this code works the modern way

...

Quite soon we’re going to learn functions (a way to group commands), so let’s note in advance that "use strict" can be put at the beginning of a function. Doing that enables strict mode 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 at the top of your scripts, otherwise strict mode may not be enabled.

Strict mode isn’t enabled here:

alert("some code");

// "use strict" below is ignored--it must be at 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 like "no use strict" that reverts the engine to old behavior.

Once we enter strict mode, there’s no going back.

## Browser console

When you use a [developer console](https://javascript.info/devtools) to run code, please note that it doesn’t use strict by default.

Sometimes, when use strict makes a difference, you’ll get incorrect results.

So, how to actually use strict in the console?

First, you can try to press Shift+Enter to input multiple lines, and put use strict on top, like this:

'use strict'; <Shift+Enter for a newline>

// ...your code

<Enter to run>

It works in most browsers, namely Firefox and Chrome.

If it doesn’t, e.g. in an old browser, there’s an ugly, but reliable way to ensure use strict. Put it inside this kind of wrapper:

(function() {

'use strict';

// ...your code here...

})()

## Should we “use strict”?

The question may sound obvious, but it’s not so.

One could recommend to start scripts with "use strict"… But you know what’s cool?

Modern JavaScript supports “classes” and “modules” – advanced language structures (we’ll surely get to them), that enable use strict automatically. So we don’t need to add the "use strict" directive, if we use them.

**So, for now "use strict"; is a welcome guest at the top of your scripts. Later, when your code is all in classes and modules, you may omit it.**

As of now, we’ve got to know about use strict in general.

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

# Variables

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

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, use the let keyword.

The statement below creates (in other words: declares) 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 'Hello' in the variable named message

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 combine 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 we don’t recommend it. 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 define multiple variables in this multiline style:

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 thing. 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” way.

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

## 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 in the box.

We can also change it as many times as we want:

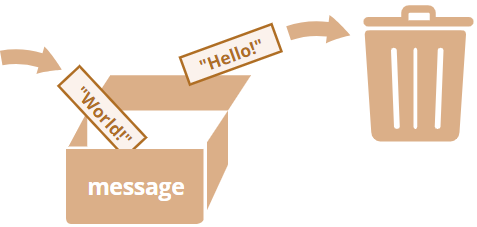
let message;

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!

**Declaring twice triggers an error**

A variable should be declared only once.

A repeated declaration of the same variable is an error:

let message = "This";

// repeated 'let' leads to an error

let message = "That"; // SyntaxError: 'message' has already been declared

So, we should declare a variable once and then refer to it without let.

**Functional languages**

It’s interesting to note that there exist [functional](https://en.wikipedia.org/wiki/Functional_programming) programming languages, like [Scala](http://www.scala-lang.org/) or [Erlang](http://www.erlang.org/) that forbid changing variable values.

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 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 you’re not planning to use it soon) is recommended to broaden the mind.

## Variable naming

There are two limitations on variable names in JavaScript:

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

Examples of valid names:

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 except first starting 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; // hyphens '-' aren't allowed in the name

**Case matters**

Variables named apple and APPLE are two different variables.

**Non-Latin 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 convention 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#reserved_words) (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical\_grammar#reserved\_words), which cannot be used as variable names because they are used by the language itself.

For example: let, class, return, and 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 using let. This still works now if we don’t put use strict in our scripts to maintain compatibility with old scripts.

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

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

alert(num); // 5

This is a bad practice and would cause an error in strict mode:

"use strict";

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

## Constants

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

const myBirthday = '18.04.1982';

Variables declared using const are called “constants”. They cannot be reassigned. An attempt to do so 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 a variable will never change, they can declare it with const to guarantee and clearly communicate that fact to everyone.

### 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.

For instance, let’s make constants for colors in so-called “web” (hexadecimal) format:

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 "#FF7F00" than 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 it normally? Let’s make that clear.

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

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

A variable name should have a clean, obvious meaning, describing the data that it stores.

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

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

Please spend time thinking about the right name for a variable before declaring it. Doing so will repay you handsomely.

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 names maximally descriptive and concise. Examples of bad names are data and value. Such names say nothing. It’s only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.
* 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 currentUser or newUser instead of currentVisitor or newManInTown.

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

**Reuse or create?**

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

As a result, their variables are like boxes into which people throw different things without changing their stickers. What’s inside the box now? Who knows? We need to come closer and check.

Such programmers save a little bit on variable declaration but lose ten times more on debugging.

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 optimize your code.

## Summary

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

let – is a modern variable declaration.

var – is an old-school variable declaration. Normally we don’t use it at all, but we’ll cover subtle differences from let in the chapter [The old "var"](https://javascript.info/var), just in case you need them.

const – is like let, but the value of the variable can’t be changed.

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

## Tasks

### Working with variables

importance: 2

1. Declare two variables: admin and name.
2. Assign the value "John" to name.
3. Copy the value from name to admin.
4. Show the value of admin using alert (must output “John”).

**Solution**

In the code below, each line corresponds to the item in the task list.

let admin, name; // can declare two variables at once

name = "John";

admin = name;

alert( admin ); // "John"

### Giving the right name

importance: 3

1. Create a variable with the name of our planet. How would you name such a variable?
2. Create a variable to store the name of a current visitor to a website. How would you name that variable?

**Solution**

The variable for our planet

That’s simple:

let ourPlanetName = "Earth";

Note, we could use a shorter name planet, but it might not be obvious what planet it refers to. It’s nice to be more verbose. At least until the variable isNotTooLong.

The name of the current visitor

let currentUserName = "John";

Again, we could shorten that to userName if we know for sure that the user is current.

Modern editors and autocomplete make long variable names easy to write. Don’t save on them. A name with 3 words in it is fine.

### Uppercase const?

importance: 4

Examine the following code:

const birthday = '18.04.1982';

const age = someCode(birthday);

Here we have a constant birthday for the date, and also the age constant.

The age is calculated from birthday using someCode(), which means a function call that we didn’t explain yet (we will soon!), but the details don’t matter here, the point is that age is calculated somehow based on the birthday.

Would it be right to use upper case for birthday? For age? Or even for both?

const BIRTHDAY = '18.04.1982'; // make birthday uppercase?

const AGE = someCode(BIRTHDAY); // make age uppercase?

**Solution**

We generally use upper case for constants that are “hard-coded”. Or, in other words, when the value is known prior to execution and directly written into the code.

In this code, birthday is exactly like that. So we could use the upper case for it.

In contrast, age is evaluated in run-time. Today we have one age, a year after we’ll have another one. It is constant in a sense that it does not change through the code execution. But it is a bit “less of a constant” than birthday: it is calculated, so we should keep the lower case for it.

# Data types

A value in JavaScript is always of a certain type. For example, a string or a number.

There are eight basic data types in JavaScript. Here, we’ll cover them in general and in the next chapters we’ll talk about each of them in detail.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

// no error

let message = "hello";

message = 123456;

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

## Number

let n = 123;

n = 12.345;

The number type represents both 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 this data type: Infinity, -Infinity and 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 reference it 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 mathematical operation on NaN returns NaN:

alert( NaN + 1 ); // NaN

alert( 3 \* NaN ); // NaN

alert( "not a number" / 2 - 1 ); // NaN

So, if there’s a NaN somewhere in a mathematical expression, it propagates to the whole result (there’s only one exception to that: NaN \*\* 0 is 1).

**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 the common sense of this word.

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

## BigInt

In JavaScript, the “number” type cannot safely represent integer values larger than (253-1) (that’s 9007199254740991), or less than -(253-1) for negatives.

To be really precise, the “number” type can store larger integers (up to 1.7976931348623157 \* 10308), but outside of the safe integer range ±(253-1) there’ll be a precision error, because not all digits fit into the fixed 64-bit storage. So an “approximate” value may be stored.

For example, these two numbers (right above the safe range) are the same:

console.log(9007199254740991 + 1); // 9007199254740992

console.log(9007199254740991 + 2); // 9007199254740992

So to say, all odd integers greater than (253-1) can’t be stored at all in the “number” type.

For most purposes ±(253-1) range is quite enough, but sometimes we need the entire range of really big integers, e.g. for cryptography or microsecond-precision timestamps.

BigInt type was recently added to the language to represent integers of arbitrary length.

A BigInt value is created by appending n to the end of an integer:

// the "n" at the end means it's a BigInt

const bigInt = 1234567890123456789012345678901234567890n;

As BigInt numbers are rarely needed, we don’t cover them here, but devoted them a separate chapter [BigInt](https://javascript.info/bigint). Read it when you need such big numbers.

**Compatibility issues**

Right now, BigInt is supported in Firefox/Chrome/Edge/Safari, but not in IE.

## String

A string in JavaScript must be surrounded by quotes.

let str = "Hello";

let str2 = 'Single quotes are ok too';

let phrase = `can embed another ${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 practically 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 in 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 don’t have this embedding functionality!

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 called “char”.

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

## 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")

## The “null” value

The special null value does not belong to any of the types 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 represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown.

## The “undefined” value

The special value undefined also 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 undefined:

let age;

alert(age); // shows "undefined"

Technically, it is possible to explicitly assign undefined to a variable:

let age = 100;

// change the value to undefined

age = undefined;

alert(age); // "undefined"

…But we don’t recommend doing that. Normally, one uses null to assign an “empty” or “unknown” value to a variable, while undefined is reserved as a default initial value for unassigned things.

## 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.

Being that important, objects deserve a special treatment. We’ll deal with them later in the chapter [Objects](https://javascript.info/object), after we learn more about primitives.

The symbol type is used to create unique identifiers for objects. We have to mention it here for the sake of completeness, but also postpone the details till we know objects.

## The typeof operator

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 do a quick check.

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

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

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 explanation:

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 an officially recognized error in typeof, coming from very early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own. The behavior of typeof is wrong here.
3. The result of typeof alert is "function", because alert is a function. We’ll study functions in the next chapters where we’ll also see that there’s no special “function” type in JavaScript. Functions belong to the object type. But typeof treats them differently, returning "function". That also comes from the early days of JavaScript. Technically, such behavior isn’t correct, but can be convenient in practice.

**The typeof(x) syntax**

You may also come across another syntax: typeof(x). It’s the same as typeof x.

To put it clear: typeof is an operator, not a function. The parentheses here aren’t a part of typeof. It’s the kind of parentheses used for mathematical grouping.

Usually, such parentheses contain a mathematical expression, such as (2 + 2), but here they contain only one argument (x). Syntactically, they allow to avoid a space between the typeof operator and its argument, and some people like it.

Some people prefer typeof(x), although the typeof x syntax is much more common.

## Summary

There are 8 basic data types in JavaScript.

* Seven primitive data types:
  + number for numbers of any kind: integer or floating-point, integers are limited by ±(253-1).
  + bigint for integer numbers of arbitrary length.
  + string for strings. A string may have zero 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.
  + symbol for unique identifiers.
* And one non-primitive data type:
  + object for more complex data structures.

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

* Usually used as typeof x, but typeof(x) is also possible.
* Returns a string with the name of the type, like "string".
* For null returns "object" – this is an error in the language, it’s not actually an object.

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

## Tasks

### String quotes

importance: 5

What is the output of the script?

let name = "Ilya";

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

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

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

**Solution**

Backticks embed the expression inside ${...} into the string.

let name = "Ilya";

// the expression is a number 1

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

// the expression is a string "name"

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

// the expression is a variable, embed it

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

# Comparisons

We know many comparison operators from maths.

In JavaScript they are written like this:

* Greater/less than: a > b, a < b.
* Greater/less than or equals: a >= b, a <= b.
* Equals: a == b, please note the double equality sign == means the equality test, while a single one a = b means an assignment.
* Not equals: In maths the notation is ≠, but in JavaScript it’s written as a != b.

## Boolean is the result

All comparison operators return a boolean value:

* true – means “yes”, “correct” or “the truth”.
* false – means “no”, “wrong” or “not the truth”.

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

To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order.

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 the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string’s, then the first string is greater (or less) than the second. We’re done.
3. Otherwise, if both strings’ first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

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

The second comparison 'Glow' and 'Glee' needs more steps as strings are compared character-by-character:

1. G is the same as G.
2. l is the same as l.
3. 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 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? The lowercase "a". Why? Because the lowercase character has a greater index in the internal encoding table JavaScript uses (Unicode). We’ll get back to specific details and consequences of this in the chapter [Strings](https://javascript.info/string).

## Comparison of different types

When comparing values of different types, JavaScript converts the values 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.

For example:

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, this result is quite normal. An equality check converts values using the numeric conversion (hence "0" becomes 0), while the explicit Boolean conversion uses another set of rules.

## Strict equality

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

alert( 0 == false ); // true

The same thing happens with an empty string:

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

This happens because operands of different types are converted to numbers 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 is also a “strict non-equality” operator !== analogous to !=.

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

## Comparison with null and undefined

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

**For a strict equality check ===**

These values are different, because each of them is a different type.

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 **< > <= >=**

null/undefined are converted to numbers: null becomes 0, while undefined becomes NaN.

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

### 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

Mathematically, that’s strange. The last result states that "null is greater than or equal to zero", so in one of the comparisons above it must be true, but they are both false.

The reason is that an equality check == and comparisons > < >= <= work differently. Comparisons convert null to a number, treating 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 is defined such that, without any conversions, they equal each other and don’t equal anything else. That’s why (2) null == 0 is false.

### An incomparable undefined

The value undefined shouldn’t be compared to other values:

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

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

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

Why does it dislike zero so much? Always false!

We get 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, undefined, and no other value.

### Avoid problems

Why did we go over 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 avoid problems with them:

* 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’re really sure of what you’re doing. If a variable can have these values, check for them separately.

## Summary

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

## Tasks

### Comparisons

importance: 5

What will be the result for these expressions?

5 > 4

"apple" > "pineapple"

"2" > "12"

undefined == null

undefined === null

null == "\n0\n"

null === +"\n0\n"

**Solution**

5 > 4 → true

"apple" > "pineapple" → false

"2" > "12" → true

undefined == null → true

undefined === null → false

null == "\n0\n" → false

null === +"\n0\n" → false

Some of the reasons:

1. Obviously, true.
2. Dictionary comparison, hence false. "a" is smaller than "p".
3. Again, dictionary comparison, first char "2" is greater than the first char "1".
4. Values null and undefined equal each other only.
5. Strict equality is strict. Different types from both sides lead to false.
6. Similar to (4), null only equals undefined.
7. Strict equality of different types.

# Conditional branching: if, '?'

Sometimes, we need to perform different actions based on different conditions.

To do that, we can use the if statement and the conditional operator ?, that’s also called a “question mark” operator.

## The “if” statement

The if(...) statement evaluates a condition in parentheses and, if the result is true, executes a block of 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 we want to execute more than one statement, we have to wrap our code block inside curly braces:

if (year == 2015) {

alert( "That's correct!" );

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

}

We recommend wrapping your code block with curly braces {} every time you use an if statement, even if there is only one statement to execute. Doing so improves readability.

## Boolean conversion

The if (…) statement evaluates the expression in its parentheses and converts the result to a boolean.

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 all 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 – it always will:

if (1) { // 1 is truthy

...

}

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

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 falsy.

For example:

let year = prompt('In which year was the 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. The else if clause lets us do that.

For example:

let year = prompt('In which year was the 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 that is falsy, it goes to the next condition year > 2015. If that is also falsy, it shows the last alert.

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

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

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

For instance:

let accessAllowed;

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

if (age > 18) {

accessAllowed = true;

} else {

accessAllowed = false;

}

alert(accessAllowed);

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

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

The syntax is:

let result = condition ? value1 : value2;

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

For example:

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

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

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

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

// (no need to wrap it into parentheses)

let accessAllowed = age > 18 ? true : false;

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

**Please note:**

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

// the same

let accessAllowed = age > 18;

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

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

For instance:

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

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

(age < 18) ? 'Hello!' :

(age < 100) ? 'Greetings!' :

'What an unusual age!';

alert( message );

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

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

Here’s how this looks using if..else:

if (age < 3) {

message = 'Hi, baby!';

} else if (age < 18) {

message = 'Hello!';

} else if (age < 100) {

message = 'Greetings!';

} else {

message = 'What an unusual age!';

}

## [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 expression after the ? gets executed and shows an alert.

We don’t assign a result to a variable here. Instead, we execute different code depending on the condition.

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

The notation is shorter than the equivalent if statement, which appeals to some programmers. But it is less readable.

Here is the same code using if for comparison:

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

if (company == 'Netscape') {

alert('Right!');

} else {

alert('Wrong.');

}

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

The purpose of the question mark operator ? is to return one value or another depending on its condition. Please use it for exactly that. Use if when you need to execute different branches of 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**

**Yes, it will.**

Any string except an empty one (and "0" is not empty) becomes true in the logical context.

We can run and check:

if ("0") {

alert( 'Hello' );

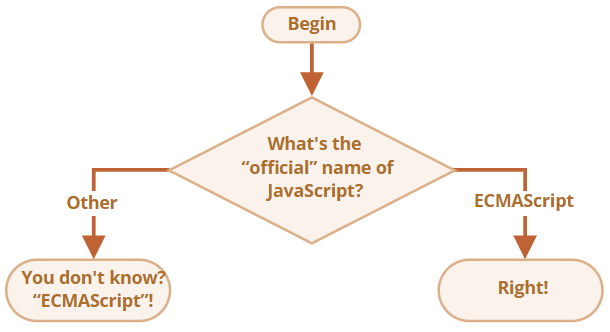
}

### [The name of JavaScript](https://javascript.info/ifelse#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: “You don’t know? ECMAScript!”



**Solution**

<!DOCTYPE html>

<html>

<body>

<script>

'use strict';

let value = prompt('What is the "official" name of JavaScript?', '');

if (value == 'ECMAScript') {

alert('Right!');

} else {

alert("You don't know? ECMAScript!");

}

</script>

</body>

</html>

### [Show the sign](https://javascript.info/ifelse#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.

**Solution**

let value = prompt('Type a number', 0);

if (value > 0) {

alert( 1 );

} else if (value < 0) {

alert( -1 );

} else {

alert( 0 );

}

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

importance: 5

Rewrite this if using the conditional operator '?':

let result;

if (a + b < 4) {

result = 'Below';

} else {

result = 'Over';

}

**Solution**

let result = (a + b < 4) ? 'Below' : 'Over';

### [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 = '';

}

**Solution**

let message = (login == 'Employee') ? 'Hello' :

(login == 'Director') ? 'Greetings' :

(login == '') ? 'No login' :

'';

# Logical operators

There are four logical operators in JavaScript: || (OR), && (AND), ! (NOT), ?? (Nullish Coalescing). Here we cover the first three, the ?? operator is in the next article.

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

Let’s see the details.

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

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

result = a || b;

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

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

There are four possible logical combinations:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

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

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

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

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

alert( 'truthy!' );

}

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

For example:

let hour = 9;

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

alert( 'The office is closed.' );

}

We can pass more conditions:

let hour = 12;

let isWeekend = true;

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

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

}

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

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

The extended algorithm works as follows.

Given multiple OR’ed values:

result = value1 || value2 || value3;

The OR || operator does the following:

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

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

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

For instance:

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

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

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

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

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

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

For instance, we have firstName, lastName and nickName variables, all optional (i.e. can be undefined or have falsy values).

Let’s use OR || to choose the one that has the data and show it (or "Anonymous" if nothing set):

let firstName = "";

let lastName = "";

let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder

If all variables were falsy, "Anonymous" would show up.

1. **Short-circuit evaluation.**

Another feature of OR || operator is the so-called “short-circuit” evaluation.

It means that || processes its arguments until the first truthy value is reached, and then the value is returned immediately, without even touching the other argument.

The importance of this feature becomes obvious if an operand isn’t just a value, but an expression with a side effect, such as a variable assignment or a function call.

In the example below, only the second message is printed:

true || alert("not printed");

false || alert("printed");

In the first line, the OR || operator stops the evaluation immediately upon seeing true, so the alert isn’t run.

Sometimes, people use this feature to execute commands only if the condition on the left part is falsy.

## [&& (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( 'The time is 12:30' );

}

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

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

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

}

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

Given multiple AND’ed values:

result = value1 && value2 && value3;

The AND && operator does the following:

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

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

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

Examples:

// if the first operand is truthy,

// AND returns the second operand:

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

// if the first operand is falsy,

// AND returns it. The second operand is ignored

alert( null && 5 ); // null

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

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

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

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

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

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

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

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

**Don’t replace if with || or &&**

Sometimes, people use the AND && operator as a "shorter way to write 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!' );

Although, the variant with && appears shorter, if is more obvious and tends to be a little bit more readable. So we recommend using every construct for its purpose: use if if we want if and use && if we want AND.

## [! (NOT)](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 the inverse value.

For instance:

alert( !true ); // false

alert( !0 ); // true

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

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

alert( !!null ); // false

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

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

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

alert( Boolean(null) ); // false

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

## [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 is the code below going to output?

alert( null || 2 || undefined );

**Solution**

The answer is 2, that’s the first truthy value.

alert( null || 2 || undefined );

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

importance: 3

What will the code below output?

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

**Solution**

The answer: first 1, then 2.

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

The call to alert does not return a value. Or, in other words, it returns undefined.

1. The first OR || evaluates its left operand alert(1). That shows the first message with 1.
2. The alert returns undefined, so OR goes on to the second operand searching for a truthy value.
3. The second operand 2 is truthy, so the execution is halted, 2 is returned and then shown by the outer alert.

There will be no 3, because the evaluation does not reach alert(3).

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

importance: 5

What is this code going to show?

alert( 1 && null && 2 );

**Solution**

The answer: null, because it’s the first falsy value from the list.

alert(1 && null && 2);

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

importance: 3

What will this code show?

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

**Solution**

The answer: 1, and then undefined.

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

The call to alert returns undefined (it just shows a message, so there’s no meaningful return).

Because of that, && evaluates the left operand (outputs 1), and immediately stops, because undefined is a falsy value. And && looks for a falsy value and returns it, so it’s done.

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

importance: 5

What will the result be?

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

**Solution**

The answer: 3.

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

The precedence of AND && is higher than ||, so it executes first.

The result of 2 && 3 = 3, so the expression becomes:

null || 3 || 4

Now the result is the first truthy value: 3.

### [Check the range between](https://javascript.info/logical-operators#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**

if (age >= 14 && age <= 90)

### [Check the range outside](https://javascript.info/logical-operators#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**

The first variant:

if (!(age >= 14 && age <= 90))

The second variant:

if (age < 14 || age > 90)

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

importance: 5

Which of these alerts are going to execute?

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

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

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

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

**Solution**

The answer: the first and the third will execute.

Details:

// Runs.

// The result of -1 || 0 = -1, truthy

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

// Doesn't run

// -1 && 0 = 0, falsy

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

// Executes

// Operator && has a higher precedence than ||

// so -1 && 1 executes first, giving us the chain:

// null || -1 && 1 -> null || 1 -> 1

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

### [Check the login](https://javascript.info/logical-operators#check-the-login)

importance: 3

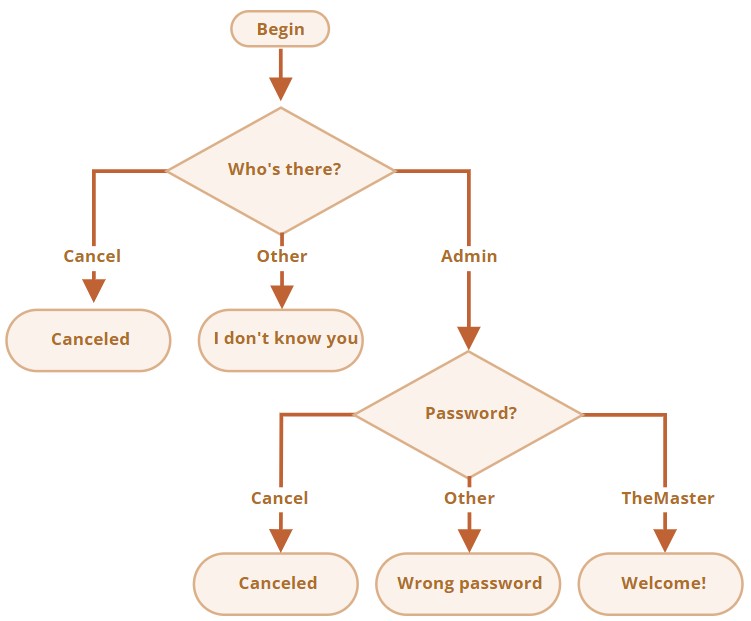
Write the code which asks for a login with prompt.

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.

**Solution**

let userName = prompt("Who's there?", '');

if (userName === 'Admin') {

let pass = prompt('Password?', '');

if (pass === 'TheMaster') {

alert( 'Welcome!' );

} else if (pass === '' || pass === null) {

alert( 'Canceled' );

} else {

alert( 'Wrong password' );

}

} else if (userName === '' || userName === null) {

alert( 'Canceled' );

} else {

alert( "I don't know you" );

}

Note the vertical indents inside the if blocks. They are technically not required but make the code more readable.

# Loops: while and for

We often need to repeat actions.

For example, outputting goods from a list one after another or just running the same code for each number from 1 to 10.

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

**The for…of and for…in loops**

A small announcement for advanced readers.

This article covers only basic loops: while, do..while and for(..;..;..).

If you came to this article searching for other types of loops, here are the pointers:

* See [for…in](https://javascript.info/object#forin) to loop over object properties.
* See [for…of](https://javascript.info/array#loops) and [iterables](https://javascript.info/iterable) for looping over arrays and iterable objects.

Otherwise, please read on.

## [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 truthy, 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 i++ was missing from the example above, the loop would repeat (in theory) forever. In practice, the browser provides ways to stop such loops, and in server-side JavaScript, we can kill the process.

Any expression or variable can be a loop condition, not just comparisons: the condition is evaluated and converted to a boolean by while.

For instance, a shorter way to write while (i != 0) is while (i):

let i = 3;

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

alert( i );

i--;

}

**Curly braces are not required for a single-line body**

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

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 should only be used 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 more complex, but it’s also the most commonly used loop.

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 | let i = 0 | Executes once upon entering the loop. |
| condition | i < 3 | Checked before every loop iteration. If false, the loop stops. |
| body | alert(i) | Runs again and again while the condition is truthy. |
| step | i++ | Executes after the body on each iteration. |

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)

→ ...

That is, begin executes once, and then it iterates: after each condition test, body and step are executed.

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

Here’s exactly what 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. This is 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 could 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++ );

}

This makes the loop identical to while (i < 3).

We can actually remove everything, creating an infinite loop:

for (;;) {

// repeats without limits

}

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

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

Normally, a loop exits when its condition becomes falsy.

But we can force the exit at any time using the special break directive.

For example, the loop below asks the user for a series of numbers, “breaking” 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 control to the first line after the loop. Namely, alert.

The combination “infinite loop + break as needed” is great for situations when a loop’s condition must be checked not in the beginning or end of the loop, but in the middle or even in several places of its 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 with the current iteration and would like to move on to the next one.

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 executing the body and passes control to the next iteration of for (with the next number). So the alert is only called for odd values.

**The continue directive helps decrease nesting**

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, this is identical to the example above. Surely, we can just wrap the code in an if block instead of using continue.

But as a side effect, this created one more level of nesting (the alert call inside the curly braces). If the code inside of 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 aren’t allowed 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 isn't allowed here

…it stops working: there’s a syntax error.

This is just another reason not to use the 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 the coordinates (i, j) from (0,0) to (2,2):

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

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

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

// what if we 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 below breaks out to the label:

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, code execution jumps to the next iteration of the labeled loop.

**Labels do not allow to “jump” anywhere**

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

For example, it is impossible to do this:

break label; // jump to the label below (doesn't work)

label: for (...)

A break directive must be inside a code block. Technically, any labelled code block will do, e.g.:

label: {

// ...

break label; // works

// ...

}

…Although, 99.9% of the time break is used inside loops, as we’ve seen in the examples above.

A continue is only possible from inside a loop.

## [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 in the current iteration and would like to forward to the next one, we can use the continue directive.

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

## [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**

The answer: 1.

let i = 3;

while (i) {

alert( i-- );

}

Every loop iteration decreases i by 1. The check while(i) stops the loop when i = 0.

Hence, the steps of the loop form the following sequence (“loop unrolled”):

let i = 3;

alert(i--); // shows 3, decreases i to 2

alert(i--) // shows 2, decreases i to 1

alert(i--) // shows 1, decreases i to 0

// done, while(i) check stops the loop

### [Which values does the while loop show?](https://javascript.info/while-for#which-values-does-the-while-loop-show)

importance: 4

For every loop iteration, write down which value it outputs and then compare it with the solution.

Both loops alert the same values, or not?

1. The prefix form ++i:

let i = 0;

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

1. The postfix form i++

let i = 0;

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

**Solution**

The task demonstrates how postfix/prefix forms can lead to different results when used in comparisons.

1. **From 1 to 4**

let i = 0;

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

The first value is i = 1, because ++i first increments i and then returns the new value. So the first comparison is 1 < 5 and the alert shows 1.

Then follow 2, 3, 4… – the values show up one after another. The comparison always uses the incremented value, because ++ is before the variable.

Finally, i = 4 is incremented to 5, the comparison while(5 < 5) fails, and the loop stops. So 5 is not shown.

1. **From 1 to 5**

let i = 0;

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

The first value is again i = 1. The postfix form of i++ increments i and then returns the old value, so the comparison i++ < 5 will use i = 0 (contrary to ++i < 5).

But the alert call is separate. It’s another statement which executes after the increment and the comparison. So it gets the current i = 1.

Then follow 2, 3, 4…

Let’s stop on i = 4. The prefix form ++i would increment it and use 5 in the comparison. But here we have the postfix form i++. So it increments i to 5, but returns the old value. Hence the comparison is actually while(4 < 5) – true, and the control goes on to alert.

The value i = 5 is the last one, because on the next step while(5 < 5) is false.

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

The answer: from 0 to 4 in both cases.

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

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

That can be easily deducted from the algorithm of for:

1. Execute once i = 0 before everything (begin).
2. Check the condition i < 5
3. If true – execute the loop body alert(i), and then i++

The increment i++ is separated from the condition check (2). That’s just another statement.

The value returned by the increment is not used here, so there’s no difference between i++ and ++i.

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

importance: 5

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

**Solution**

for (let i = 2; i <= 10; i++) {

if (i % 2 == 0) {

alert( i );

}

}

We use the “modulo” operator % to get the remainder and check for the evenness here.

### [Replace "for" with "while"](https://javascript.info/while-for#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**

let i = 0;

while (i < 3) {

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

i++;

}

### [Repeat until the input is correct](https://javascript.info/while-for#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 them 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.

**Solution**

let num;

do {

num = prompt("Enter a number greater than 100?", 0);

} while (num <= 100 && num);

The loop do..while repeats while both checks are truthy:

1. The check for num <= 100 – that is, the entered value is still not greater than 100.
2. The check && num is false when num is null or an empty string. Then the while loop stops too.

P.S. If num is null then num <= 100 is true, so without the 2nd check the loop wouldn’t stop if the user clicks CANCEL. Both checks are required.

### [Output prime numbers](https://javascript.info/while-for#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**

There are many algorithms for this task.

Let’s use a nested loop:

For each i in the interval {

check if i has a divisor from 1..i

if yes => the value is not a prime

if no => the value is a prime, show it

}

The code using a label:

let n = 10;

nextPrime:

for (let i = 2; i <= n; i++) { // for each i...

for (let j = 2; j < i; j++) { // look for a divisor..

if (i % j == 0) continue nextPrime; // not a prime, go next i

}

alert( i ); // a prime

}

There’s a lot of space to optimize it. For instance, we could look for the divisors from 2 to square root of i. But anyway, if we want to be really efficient for large intervals, we need to change the approach and rely on advanced maths and complex algorithms like [Quadratic sieve](https://en.wikipedia.org/wiki/Quadratic_sieve), [General number field sieve](https://en.wikipedia.org/wiki/General_number_field_sieve) etc.

# 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 (comma-separated, empty in the example above, we’ll see examples later) and finally the code of the function, also named “the function body”, between curly braces.

function name(parameter1, parameter2, ... parameterN) {

// body

}

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.

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).

It’s a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

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

We can pass arbitrary data to functions using parameters.

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

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

When a value is passed as a function parameter, it’s also called an argument.

In other words, to put these terms straight:

* A parameter is the variable listed inside the parentheses in the function declaration (it’s a declaration time term).
* An argument is the value that is passed to the function when it is called (it’s a call time term).

We declare functions listing their parameters, then call them passing arguments.

In the example above, one might say: "the function showMessage is declared with two parameters, then called with two arguments: from and "Hello"".

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

If a function is called, but an argument is not provided, then the corresponding 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". As the value for text isn’t passed, it becomes undefined.

We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using =:

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".

The default value also jumps in if the parameter exists, but strictly equals undefined, like this:

showMessage("Ann", undefined); // Ann: 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

}

**Evaluation of default parameters**

In JavaScript, a default parameter is evaluated every time the function is called without the respective parameter.

In the example above, anotherFunction() isn’t called at all, if the text parameter is provided.

On the other hand, it’s independently called every time when text is missing.

**Default parameters in old JavaScript code**

Several years ago, JavaScript didn’t support the syntax for default parameters. So people used other ways to specify them.

Nowadays, we can come across them in old scripts.

For example, an explicit check for undefined:

function showMessage(from, text) {

if (text === undefined) {

text = 'no text given';

}

alert( from + ": " + text );

}

…Or using the || operator:

function showMessage(from, text) {

// If the value of text is falsy, assign the default value

// this assumes that text == "" is the same as no text at all

text = text || 'no text given';

...

}

### [Alternative default parameters](https://javascript.info/function-basics" \l "alternative-default-parameters)

Sometimes it makes sense to assign default values for parameters at a later stage after the function declaration.

We can check if the parameter is passed during the function execution, by comparing it with undefined:

function showMessage(text) {

// ...

if (text === undefined) { // if the parameter is missing

text = 'empty message';

}

alert(text);

}

showMessage(); // empty message

…Or we could use the || operator:

function showMessage(text) {

// if text is undefined or otherwise falsy, set it to 'empty'

text = text || 'empty';

...

}

Modern JavaScript engines support the [nullish coalescing operator](https://javascript.info/nullish-coalescing-operator) ??, it’s better when most falsy values, such as 0, should be considered “normal”:

function showCount(count) {

// if count is undefined or null, show "unknown"

alert(count ?? "unknown");

}

showCount(0); // 0

showCount(null); // unknown

showCount(); // unknown

## [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('Do you have permission from your 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.

If we want the returned expression to wrap across multiple lines, we should start it at the same line as return. Or at least put the opening parentheses there as follows:

return (

some + long + expression

+ or +

whatever \* f(a) + f(b)

)

And it will work just as we expect it to.

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

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

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 it displays the access granted/denied message (should only perform the check and return the result).

These examples assume common meanings of prefixes. You and your team are free to agree on other meanings, but usually they’re not much different. In any case, 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 with $. The [Lodash](http://lodash.com/) library has its core function named \_.

These are exceptions. Generally function names should be concise and 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**

No difference!

In both cases, return confirm('Did parents allow you?') executes exactly when the if condition is falsy.

### [Rewrite the function using '?' or '||'](https://javascript.info/function-basics#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('Did parents allow you?');

}

}

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**

Using a question mark operator '?':

function checkAge(age) {

return (age > 18) ? true : confirm('Did parents allow you?');

}

Using OR || (the shortest variant):

function checkAge(age) {

return (age > 18) || confirm('Did parents allow you?');

}

Note that the parentheses around age > 18 are not required here. They exist for better readability.

### [Function min(a, b)](https://javascript.info/function-basics#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**

A solution using if:

function min(a, b) {

if (a < b) {

return a;

} else {

return b;

}

}

A solution with a question mark operator '?':

function min(a, b) {

return a < b ? a : b;

}

P.S. In the case of an equality a == b it does not matter what to return.

### [Function pow(x,n)](https://javascript.info/function-basics#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).

P.S. In this task the function should support only natural values of n: integers up from 1.

**Solution**

function pow(x, n) {

let result = x;

for (let i = 1; i < n; i++) {

result \*= x;

}

return result;

}

let x = prompt("x?", '');

let n = prompt("n?", '');

if (n < 1) {

alert(`Power ${n} is not supported, use a positive integer`);

} else {

alert( pow(x, n) );

}

# JavaScript – this

The “**this**” keyword refers to an object that is executing the current piece of code. It references the object that is executing the current function. If the function being referenced is a regular function, “this” references the global object. If the function that is being referenced is a method in an object, “this” references the object itself.

The JavaScript “this” keyword is one of the most widely used keywords. It can seem complex at first, but once you start using “this” keyword, everything will become clear.

## Global Scope

If you call a function from the global scope that includes “this” keyword, then “this” will always point to the window object. The following example can help to visualize this concept:

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>JavaScript This Keyword</title>

  </head>

  <body>

    <script>

      var myVar = 100;

      function printMe() {

        var myVar = 200;

        alert("myVar = " + myVar); // 200

        alert("this.myVar = " + this.myVar); // 100

      }

      printMe();

    </script>

  </body>

</html>

* We declare a variable myVar with an initial value of 100
* After that, we define a function that sends an alert with a statement to the browser window when it is called
* Whenever we access the value of myVar without “this” keyword, it gets the value defined locally within that function
* On the other hand, when we try to get the value of myVar variable using the “this” keyword, it fetches the value defined outside the function (globally)

## Object Method

In JavaScript, we can create objects of a function using the new keyword. Therefore, whenever we create an object of a function using the new keyword, then “this” will point to that particular object. We can take a look at this in more detail with the following example:

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>JavaScript This Keyword</title>

  </head>

  <body>

    <script>

      var myVar = 100;

      function printMe() {

        this.myVar = 200;

        this.display = function () {

          var myVar = 300;

          alert("myVar = " + myVar); // 300

          alert("this.myVar = " + this.myVar); // 200

        };

      }

      var obj = new printMe();

      obj.display();

    </script>

  </body>

</html>

* We declare a variable myVar and initialize it with a value of 100
* Now, when we create an object ‘obj’, it will have two properties that we defined in the code: ‘myVar’ and ‘display’—note that display is a function expression in this case
* Therefore, “this” keyword inside the display() method of the object ‘obj’ points to the value outside the scope of the display() method
* When we call the display() method, it initially shows the alert with the local value, and then an additional alert with the global value fetched using “this” keyword

# Class basic syntax

*In object-oriented programming, a*class*is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).*

Wikipedia

In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

As we already know from the chapter [Constructor, operator "new"](https://javascript.info/constructor-new), new function can help with that.

But in the modern JavaScript, there’s a more advanced “class” construct, that introduces great new features which are useful for object-oriented programming.

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

The basic syntax is:

class MyClass {

// class methods

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Then use new MyClass() to create a new object with all the listed methods.

The constructor() method is called automatically by new, so we can initialize the object there.

For example:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Usage:

let user = new User("John");

user.sayHi();

When new User("John") is called:

1. A new object is created.
2. The constructor runs with the given argument and assigns it to this.name.

…Then we can call object methods, such as user.sayHi().

**No comma between class methods**

A common pitfall for novice developers is to put a comma between class methods, which would result in a syntax error.

The notation here is not to be confused with object literals. Within the class, no commas are required.

## [What is a class?](https://javascript.info/class" \l "what-is-a-class)

So, what exactly is a class? That’s not an entirely new language-level entity, as one might think.

Let’s unveil any magic and see what a class really is. That’ll help in understanding many complex aspects.

In JavaScript, a class is a kind of function.

Here, take a look:

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

// proof: User is a function

alert(typeof User); // function

What class User {...} construct really does is:

1. Creates a function named User, that becomes the result of the class declaration. The function code is taken from the constructor method (assumed empty if we don’t write such method).
2. Stores class methods, such as sayHi, in User.prototype.

After new User object is created, when we call its method, it’s taken from the prototype, just as described in the chapter [F.prototype](https://javascript.info/function-prototype). So, the object has access to class methods.

We can illustrate the result of class User declaration as:

Here’s the code to introspect it:

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

// class is a function

alert(typeof User); // function

// ...or, more precisely, the constructor method

alert(User === User.prototype.constructor); // true

// The methods are in User.prototype, e.g:

alert(User.prototype.sayHi); // the code of the sayHi method

// there are exactly two methods in the prototype

alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

## [Not just a syntactic sugar](https://javascript.info/class" \l "not-just-a-syntactic-sugar)

Sometimes people say that class is a “syntactic sugar” (syntax that is designed to make things easier to read, but doesn’t introduce anything new), because we could actually declare the same thing without using the class keyword at all:

// rewriting class User in pure functions

// 1. Create constructor function

function User(name) {

this.name = name;

}

// a function prototype has "constructor" property by default,

// so, we don't need to create it

// 2. Add the method to prototype

User.prototype.sayHi = function() {

alert(this.name);

};

// Usage:

let user = new User("John");

user.sayHi();

The result of this definition is about the same. So, there are indeed reasons why class can be considered a syntactic sugar to define a constructor together with its prototype methods.

Still, there are important differences.

1. First, a function created by class is labelled by a special internal property [[IsClassConstructor]]: true. So it’s not entirely the same as creating it manually.

The language checks for that property in a variety of places. For example, unlike a regular function, it must be called with new:

class User {

constructor() {}

}

alert(typeof User); // function

User(); // Error: Class constructor User cannot be invoked without 'new'

Also, a string representation of a class constructor in most JavaScript engines starts with the “class…”

class User {

constructor() {}

}

alert(User); // class User { ... }

There are other differences, we’ll see them soon.

1. Class methods are non-enumerable. A class definition sets enumerable flag to false for all methods in the "prototype".

That’s good, because if we for..in over an object, we usually don’t want its class methods.

1. Classes always use strict. All code inside the class construct is automatically in strict mode.

Besides, class syntax brings many other features that we’ll explore later.

## [Class Expression](https://javascript.info/class" \l "class-expression)

Just like functions, classes can be defined inside another expression, passed around, returned, assigned, etc.

Here’s an example of a class expression:

let User = class {

sayHi() {

alert("Hello");

}

};

Similar to Named Function Expressions, class expressions may have a name.

If a class expression has a name, it’s visible inside the class only:

// "Named Class Expression"

// (no such term in the spec, but that's similar to Named Function Expression)

let User = class MyClass {

sayHi() {

alert(MyClass); // MyClass name is visible only inside the class

}

};

new User().sayHi(); // works, shows MyClass definition

alert(MyClass); // error, MyClass name isn't visible outside of the class

We can even make classes dynamically “on-demand”, like this:

function makeClass(phrase) {

// declare a class and return it

return class {

sayHi() {

alert(phrase);

}

};

}

// Create a new class

let User = makeClass("Hello");

new User().sayHi(); // Hello

## [Getters/setters](https://javascript.info/class" \l "getters-setters)

Just like literal objects, classes may include getters/setters, computed properties etc.

Here’s an example for user.name implemented using get/set:

class User {

constructor(name) {

// invokes the setter

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Name is too short.");

return;

}

this.\_name = value;

}

}

let user = new User("John");

alert(user.name); // John

user = new User(""); // Name is too short.

Technically, such class declaration works by creating getters and setters in User.prototype.

## [Computed names […]](https://javascript.info/class" \l "computed-names)

Here’s an example with a computed method name using brackets [...]:

class User {

['say' + 'Hi']() {

alert("Hello");

}

}

new User().sayHi();

Such features are easy to remember, as they resemble that of literal objects.

## [Class fields](https://javascript.info/class" \l "class-fields)

**Old browsers may need a polyfill**

Class fields are a recent addition to the language.

Previously, our classes only had methods.

“Class fields” is a syntax that allows to add any properties.

For instance, let’s add name property to class User:

class User {

name = "John";

sayHi() {

alert(`Hello, ${this.name}!`);

}

}

new User().sayHi(); // Hello, John!

So, we just write " = " in the declaration, and that’s it.

The important difference of class fields is that they are set on individual objects, not User.prototype:

class User {

name = "John";

}

let user = new User();

alert(user.name); // John

alert(User.prototype.name); // undefined

We can also assign values using more complex expressions and function calls:

class User {

name = prompt("Name, please?", "John");

}

let user = new User();

alert(user.name); // John

### [Making bound methods with class fields](https://javascript.info/class" \l "making-bound-methods-with-class-fields)

As demonstrated in the chapter [Function binding](https://javascript.info/bind) functions in JavaScript have a dynamic this. It depends on the context of the call.

So, if an object method is passed around and called in another context, this won’t be a reference to its object any more.

For instance, this code will show undefined:

class Button {

constructor(value) {

this.value = value;

}

click() {

alert(this.value);

}

}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined

The problem is called "losing this".

There are two approaches to fixing it, as discussed in the chapter [Function binding](https://javascript.info/bind):

1. Pass a wrapper-function, such as setTimeout(() => button.click(), 1000).
2. Bind the method to object, e.g., in the constructor.

Class fields provide another, quite elegant syntax:

class Button {

constructor(value) {

this.value = value;

}

click = () => {

alert(this.value);

}

}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello

The class field click = () => {...} is created on a per-object basis, there’s a separate function for each Button object, with this inside it referencing that object. We can pass button.click around anywhere, and the value of this will always be correct.

That’s especially useful in browser environment, for event listeners.

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

The basic class syntax looks like this:

class MyClass {

prop = value; // property

constructor(...) { // constructor

// ...

}

method(...) {} // method

get something(...) {} // getter method

set something(...) {} // setter method

[Symbol.iterator]() {} // method with computed name (symbol here)

// ...

}

MyClass is technically a function (the one that we provide as constructor), while methods, getters and setters are written to MyClass.prototype.

In the next chapters we’ll learn more about classes, including inheritance and other features.

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

### [Rewrite to class](https://javascript.info/class" \l "rewrite-to-class)

importance: 5

The Clock class (see the sandbox) is written in functional style. Rewrite it in the “class” syntax.

P.S. The clock ticks in the console, open it to see.

**Solution**

class Clock {

constructor({ template }) {

this.template = template;

}

render() {

let date = new Date();

let hours = date.getHours();

if (hours < 10) hours = '0' + hours;

let mins = date.getMinutes();

if (mins < 10) mins = '0' + mins;

let secs = date.getSeconds();

if (secs < 10) secs = '0' + secs;

let output = this.template

.replace('h', hours)

.replace('m', mins)

.replace('s', secs);

console.log(output);

}

stop() {

clearInterval(this.timer);

}

start() {

this.render();

this.timer = setInterval(() => this.render(), 1000);

}

}

let clock = new Clock({template: 'h:m:s'});

clock.start();

# Class inheritance

Class inheritance is a way for one class to extend another class.

So we can create new functionality on top of the existing.

## [The “extends” keyword](https://javascript.info/class-inheritance" \l "the-extends-keyword)

Let’s say we have class Animal:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} runs with speed ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} stands still.`);

}

}

let animal = new Animal("My animal");

Here’s how we can represent animal object and Animal class graphically:

Diagram

Description automatically generated

…And we would like to create another class Rabbit.

As rabbits are animals, Rabbit class should be based on Animal, have access to animal methods, so that rabbits can do what “generic” animals can do.

The syntax to extend another class is: class Child extends Parent.

Let’s create class Rabbit that inherits from Animal:

class Rabbit extends Animal {

hide() {

alert(`${this.name} hides!`);

}

}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.

rabbit.hide(); // White Rabbit hides!

Object of Rabbit class have access both to Rabbit methods, such as rabbit.hide(), and also to Animal methods, such as rabbit.run().

Internally, extends keyword works using the good old prototype mechanics. It sets Rabbit.prototype.[[Prototype]] to Animal.prototype. So, if a method is not found in Rabbit.prototype, JavaScript takes it from Animal.prototype.

Diagram

Description automatically generated

For instance, to find rabbit.run method, the engine checks (bottom-up on the picture):

1. The rabbit object (has no run).
2. Its prototype, that is Rabbit.prototype (has hide, but not run).
3. Its prototype, that is (due to extends) Animal.prototype, that finally has the run method.

As we can recall from the chapter [Native prototypes](https://javascript.info/native-prototypes), JavaScript itself uses prototypal inheritance for built-in objects. E.g. Date.prototype.[[Prototype]] is Object.prototype. That’s why dates have access to generic object methods.

**Any expression is allowed after extends**

Class syntax allows to specify not just a class, but any expression after extends.

For instance, a function call that generates the parent class:

function f(phrase) {

return class {

sayHi() { alert(phrase); }

};

}

class User extends f("Hello") {}

new User().sayHi(); // Hello

Here class User inherits from the result of f("Hello").

That may be useful for advanced programming patterns when we use functions to generate classes depending on many conditions and can inherit from them.

## [Overriding a method](https://javascript.info/class-inheritance" \l "overriding-a-method)

Now let’s move forward and override a method. By default, all methods that are not specified in class Rabbit are taken directly “as is” from class Animal.

But if we specify our own method in Rabbit, such as stop() then it will be used instead:

class Rabbit extends Animal {

stop() {

// ...now this will be used for rabbit.stop()

// instead of stop() from class Animal

}

}

Usually, however, we don’t want to totally replace a parent method, but rather to build on top of it to tweak or extend its functionality. We do something in our method, but call the parent method before/after it or in the process.

Classes provide "super" keyword for that.

* super.method(...) to call a parent method.
* super(...) to call a parent constructor (inside our constructor only).

For instance, let our rabbit autohide when stopped:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} runs with speed ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} stands still.`);

}

}

class Rabbit extends Animal {

hide() {

alert(`${this.name} hides!`);

}

stop() {

super.stop(); // call parent stop

this.hide(); // and then hide

}

}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.

rabbit.stop(); // White Rabbit stands still. White Rabbit hides!

Now Rabbit has the stop method that calls the parent super.stop() in the process.

**Arrow functions have no super**

As was mentioned in the chapter [Arrow functions revisited](https://javascript.info/arrow-functions), arrow functions do not have super.

If accessed, it’s taken from the outer function. For instance:

class Rabbit extends Animal {

stop() {

setTimeout(() => super.stop(), 1000); // call parent stop after 1sec

}

}

The super in the arrow function is the same as in stop(), so it works as intended. If we specified a “regular” function here, there would be an error:

// Unexpected super

setTimeout(function() { super.stop() }, 1000);

## [Overriding constructor](https://javascript.info/class-inheritance" \l "overriding-constructor)

With constructors it gets a little bit tricky.

Until now, Rabbit did not have its own constructor.

According to the [specification](https://tc39.github.io/ecma262/#sec-runtime-semantics-classdefinitionevaluation), if a class extends another class and has no constructor, then the following “empty” constructor is generated:

class Rabbit extends Animal {

// generated for extending classes without own constructors

constructor(...args) {

super(...args);

}

}

As we can see, it basically calls the parent constructor passing it all the arguments. That happens if we don’t write a constructor of our own.

Now let’s add a custom constructor to Rabbit. It will specify the earLength in addition to name:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

this.speed = 0;

this.name = name;

this.earLength = earLength;

}

// ...

}

// Doesn't work!

let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.

Whoops! We’ve got an error. Now we can’t create rabbits. What went wrong?

The short answer is:

* **Constructors in inheriting classes must call super(...), and (!) do it before using this.**

…But why? What’s going on here? Indeed, the requirement seems strange.

Of course, there’s an explanation. Let’s get into details, so you’ll really understand what’s going on.

In JavaScript, there’s a distinction between a constructor function of an inheriting class (so-called “derived constructor”) and other functions. A derived constructor has a special internal property [[ConstructorKind]]:"derived". That’s a special internal label.

That label affects its behavior with new.

* When a regular function is executed with new, it creates an empty object and assigns it to this.
* But when a derived constructor runs, it doesn’t do this. It expects the parent constructor to do this job.

So a derived constructor must call super in order to execute its parent (base) constructor, otherwise the object for this won’t be created. And we’ll get an error.

For the Rabbit constructor to work, it needs to call super() before using this, like here:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

super(name);

this.earLength = earLength;

}

// ...

}

// now fine

let rabbit = new Rabbit("White Rabbit", 10);

alert(rabbit.name); // White Rabbit

alert(rabbit.earLength); // 10

### [Overriding class fields: a tricky note](https://javascript.info/class-inheritance" \l "overriding-class-fields-a-tricky-note)

**Advanced note**

This note assumes you have a certain experience with classes, maybe in other programming languages.

It provides better insight into the language and also explains the behavior that might be a source of bugs (but not very often).

If you find it difficult to understand, just go on, continue reading, then return to it some time later.

We can override not only methods, but also class fields.

Although, there’s a tricky behavior when we access an overridden field in parent constructor, quite different from most other programming languages.

Consider this example:

class Animal {

name = 'animal';

constructor() {

alert(this.name); // (\*)

}

}

class Rabbit extends Animal {

name = 'rabbit';

}

new Animal(); // animal

new Rabbit(); // animal

Here, class Rabbit extends Animal and overrides the name field with its own value.

There’s no own constructor in Rabbit, so Animal constructor is called.

What’s interesting is that in both cases: new Animal() and new Rabbit(), the alert in the line (\*) shows animal.

**In other words, the parent constructor always uses its own field value, not the overridden one.**

What’s odd about it?

If it’s not clear yet, please compare with methods.

Here’s the same code, but instead of this.name field we call this.showName() method:

class Animal {

showName() { // instead of this.name = 'animal'

alert('animal');

}

constructor() {

this.showName(); // instead of alert(this.name);

}

}

class Rabbit extends Animal {

showName() {

alert('rabbit');

}

}

new Animal(); // animal

new Rabbit(); // rabbit

Please note: now the output is different.

And that’s what we naturally expect. When the parent constructor is called in the derived class, it uses the overridden method.

…But for class fields it’s not so. As said, the parent constructor always uses the parent field.

Why is there a difference?

Well, the reason is the field initialization order. The class field is initialized:

* Before constructor for the base class (that doesn’t extend anything),
* Immediately after super() for the derived class.

In our case, Rabbit is the derived class. There’s no constructor() in it. As said previously, that’s the same as if there was an empty constructor with only super(...args).

So, new Rabbit() calls super(), thus executing the parent constructor, and (per the rule for derived classes) only after that its class fields are initialized. At the time of the parent constructor execution, there are no Rabbit class fields yet, that’s why Animal fields are used.

This subtle difference between fields and methods is specific to JavaScript.

Luckily, this behavior only reveals itself if an overridden field is used in the parent constructor. Then it may be difficult to understand what’s going on, so we’re explaining it here.

If it becomes a problem, one can fix it by using methods or getters/setters instead of fields.

## [Super: internals, [[HomeObject]]](https://javascript.info/class-inheritance" \l "super-internals-homeobject)

**Advanced information**

If you’re reading the tutorial for the first time – this section may be skipped.

It’s about the internal mechanisms behind inheritance and super.

Let’s get a little deeper under the hood of super. We’ll see some interesting things along the way.

First to say, from all that we’ve learned till now, it’s impossible for super to work at all!

Yeah, indeed, let’s ask ourselves, how it should technically work? When an object method runs, it gets the current object as this. If we call super.method() then, the engine needs to get the method from the prototype of the current object. But how?

The task may seem simple, but it isn’t. The engine knows the current object this, so it could get the parent method as this.\_\_proto\_\_.method. Unfortunately, such a “naive” solution won’t work.

Let’s demonstrate the problem. Without classes, using plain objects for the sake of simplicity.

You may skip this part and go below to the [[HomeObject]] subsection if you don’t want to know the details. That won’t harm. Or read on if you’re interested in understanding things in-depth.

In the example below, rabbit.\_\_proto\_\_ = animal. Now let’s try: in rabbit.eat() we’ll call animal.eat(), using this.\_\_proto\_\_:

let animal = {

name: "Animal",

eat() {

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Rabbit",

eat() {

// that's how super.eat() could presumably work

this.\_\_proto\_\_.eat.call(this); // (\*)

}

};

rabbit.eat(); // Rabbit eats.

At the line (\*) we take eat from the prototype (animal) and call it in the context of the current object. Please note that .call(this) is important here, because a simple this.\_\_proto\_\_.eat() would execute parent eat in the context of the prototype, not the current object.

And in the code above it actually works as intended: we have the correct alert.

Now let’s add one more object to the chain. We’ll see how things break:

let animal = {

name: "Animal",

eat() {

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat() {

// ...bounce around rabbit-style and call parent (animal) method

this.\_\_proto\_\_.eat.call(this); // (\*)

}

};

let longEar = {

\_\_proto\_\_: rabbit,

eat() {

// ...do something with long ears and call parent (rabbit) method

this.\_\_proto\_\_.eat.call(this); // (\*\*)

}

};

longEar.eat(); // Error: Maximum call stack size exceeded

The code doesn’t work anymore! We can see the error trying to call longEar.eat().

It may be not that obvious, but if we trace longEar.eat() call, then we can see why. In both lines (\*) and (\*\*) the value of this is the current object (longEar). That’s essential: all object methods get the current object as this, not a prototype or something.

So, in both lines (\*) and (\*\*) the value of this.\_\_proto\_\_ is exactly the same: rabbit. They both call rabbit.eat without going up the chain in the endless loop.

Diagram

Description automatically generated

Here’s the picture of what happens:

1. Inside longEar.eat(), the line (\*\*) calls rabbit.eat providing it with this=longEar.

// inside longEar.eat() we have this = longEar

this.\_\_proto\_\_.eat.call(this) // (\*\*)

// becomes

longEar.\_\_proto\_\_.eat.call(this)

// that is

rabbit.eat.call(this);

1. Then in the line (\*) of rabbit.eat, we’d like to pass the call even higher in the chain, but this=longEar, so this.\_\_proto\_\_.eat is again rabbit.eat!

// inside rabbit.eat() we also have this = longEar

this.\_\_proto\_\_.eat.call(this) // (\*)

// becomes

longEar.\_\_proto\_\_.eat.call(this)

// or (again)

rabbit.eat.call(this);

1. …So rabbit.eat calls itself in the endless loop, because it can’t ascend any further.

The problem can’t be solved by using this alone.

### [[[HomeObject]]](https://javascript.info/class-inheritance" \l "homeobject)

To provide the solution, JavaScript adds one more special internal property for functions: [[HomeObject]].

When a function is specified as a class or object method, its [[HomeObject]] property becomes that object.

Then super uses it to resolve the parent prototype and its methods.

Let’s see how it works, first with plain objects:

let animal = {

name: "Animal",

eat() { // animal.eat.[[HomeObject]] == animal

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Rabbit",

eat() { // rabbit.eat.[[HomeObject]] == rabbit

super.eat();

}

};

let longEar = {

\_\_proto\_\_: rabbit,

name: "Long Ear",

eat() { // longEar.eat.[[HomeObject]] == longEar

super.eat();

}

};

// works correctly

longEar.eat(); // Long Ear eats.

It works as intended, due to [[HomeObject]] mechanics. A method, such as longEar.eat, knows its [[HomeObject]] and takes the parent method from its prototype. Without any use of this.

### [Methods are not “free”](https://javascript.info/class-inheritance" \l "methods-are-not-free)

As we’ve known before, generally functions are “free”, not bound to objects in JavaScript. So they can be copied between objects and called with another this.

The very existence of [[HomeObject]] violates that principle, because methods remember their objects. [[HomeObject]] can’t be changed, so this bond is forever.

The only place in the language where [[HomeObject]] is used – is super. So, if a method does not use super, then we can still consider it free and copy between objects. But with super things may go wrong.

Here’s the demo of a wrong super result after copying:

let animal = {

sayHi() {

alert(`I'm an animal`);

}

};

// rabbit inherits from animal

let rabbit = {

\_\_proto\_\_: animal,

sayHi() {

super.sayHi();

}

};

let plant = {

sayHi() {

alert("I'm a plant");

}

};

// tree inherits from plant

let tree = {

\_\_proto\_\_: plant,

sayHi: rabbit.sayHi // (\*)

};

tree.sayHi(); // I'm an animal (?!?)

A call to tree.sayHi() shows “I’m an animal”. Definitely wrong.

The reason is simple:

* In the line (\*), the method tree.sayHi was copied from rabbit. Maybe we just wanted to avoid code duplication?
* Its [[HomeObject]] is rabbit, as it was created in rabbit. There’s no way to change [[HomeObject]].
* The code of tree.sayHi() has super.sayHi() inside. It goes up from rabbit and takes the method from animal.

Here’s the diagram of what happens:

Diagram

Description automatically generated with medium confidence

### [Methods, not function properties](https://javascript.info/class-inheritance" \l "methods-not-function-properties)

[[HomeObject]] is defined for methods both in classes and in plain objects. But for objects, methods must be specified exactly as method(), not as "method: function()".

The difference may be non-essential for us, but it’s important for JavaScript.

In the example below a non-method syntax is used for comparison. [[HomeObject]] property is not set and the inheritance doesn’t work:

let animal = {

eat: function() { // intentionally writing like this instead of eat() {...

// ...

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat: function() {

super.eat();

}

};

rabbit.eat(); // Error calling super (because there's no [[HomeObject]])

## [Summary](https://javascript.info/class-inheritance#summary)

1. To extend a class: class Child extends Parent:

* That means Child.prototype.\_\_proto\_\_ will be Parent.prototype, so methods are inherited.

1. When overriding a constructor:

* We must call parent constructor as super() in Child constructor before using this.

1. When overriding another method:

* We can use super.method() in a Child method to call Parent method.

1. Internals:

* Methods remember their class/object in the internal [[HomeObject]] property. That’s how super resolves parent methods.
* So it’s not safe to copy a method with super from one object to another.

Also:

* Arrow functions don’t have their own this or super, so they transparently fit into the surrounding context.

## [Tasks](https://javascript.info/class-inheritance#tasks)

### [Error creating an instance](https://javascript.info/class-inheritance" \l "error-creating-an-instance)

importance: 5

Here’s the code with Rabbit extending Animal.

Unfortunately, Rabbit objects can’t be created. What’s wrong? Fix it.

class Animal {

constructor(name) {

this.name = name;

}

}

class Rabbit extends Animal {

constructor(name) {

this.name = name;

this.created = Date.now();

}

}

let rabbit = new Rabbit("White Rabbit"); // Error: this is not defined

alert(rabbit.name);

**Solution**

That’s because the child constructor must call super().

Here’s the corrected code:

class Animal {

constructor(name) {

this.name = name;

}

}

class Rabbit extends Animal {

constructor(name) {

super(name);

this.created = Date.now();

}

}

let rabbit = new Rabbit("White Rabbit"); // ok now

alert(rabbit.name); // White Rabbit

### [Extended clock](https://javascript.info/class-inheritance#extended-clock)

importance: 5

We’ve got a Clock class. As of now, it prints the time every second.

class Clock {

constructor({ template }) {

this.template = template;

}

render() {

let date = new Date();

let hours = date.getHours();

if (hours < 10) hours = '0' + hours;

let mins = date.getMinutes();

if (mins < 10) mins = '0' + mins;

let secs = date.getSeconds();

if (secs < 10) secs = '0' + secs;

let output = this.template

.replace('h', hours)

.replace('m', mins)

.replace('s', secs);

console.log(output);

}

stop() {

clearInterval(this.timer);

}

start() {

this.render();

this.timer = setInterval(() => this.render(), 1000);

}

}

Create a new class ExtendedClock that inherits from Clock and adds the parameter precision – the number of ms between “ticks”. Should be 1000 (1 second) by default.

* Your code should be in the file extended-clock.js
* Don’t modify the original clock.js. Extend it.

**Solution**

class ExtendedClock extends Clock {

constructor(options) {

super(options);

let { precision = 1000 } = options;

this.precision = precision;

}

start() {

this.render();

this.timer = setInterval(() => this.render(), this.precision);

}

};

# JavaScript – ES6

<https://www.w3schools.com/js/js_es6.asp>

ECMAScript 2015 was the second major revision to JavaScript.

ECMAScript 2015 is also known as ES6 and ECMAScript 6.

## JavaScript let

The let keyword allows you to declare a variable with block scope.

<!DOCTYPE html>

<html>

<body>

<h2>Redeclaring a Variable Using let</h2>

<p id="demo"></p>

<script>

let x = 10;

// Here x is 10

{

let x = 2;

// Here x is 2

}

// Here x is 10

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

## JavaScript const

The const keyword allows you to declare a constant (a JavaScript variable with a constant value).

Constants are similar to let variables, except that the value cannot be changed.

<!DOCTYPE html>

<html>

<body>

<h2>Declaring a Variable Using const</h2>

<p id="demo"></p>

<script>

var x = 10;

// Here x is 10

{

const x = 2;

// Here x is 2

}

// Here x is 10

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

## Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Arrow Functions</h2>

<p>With arrow functions, you don't have to type the function keyword, the return keyword, and the curly brackets.</p>

<p>Arrow functions are not supported in IE11 or earlier.</p>

<p id="demo"></p>

<script>

const x = (x, y) => x \* y;

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always a constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Arrow Functions</h2>

<p>Arrow functions are not supported in IE11 or earlier.</p>

<p id="demo"></p>

<script>

const x = (x, y) => { return x \* y };

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

## The Spread (...) Operator

The ... operator expands an iterable (like an array) into more elements:

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Operators</h1>

<h2>The ... Operator</h2>

<p>The "spread" operator spreads elements of iterable objects:</p>

<p id="demo"></p>

<script>

const cars1 = ["Saab", "Volvo", ..."BMW"];

const cars2 = ["Fiat", "Toyota"];

const combined = [cars1, ...cars2];

document.getElementById("demo").innerHTML = combined;

</script>

</body>

</html>

The ... operator can be used to expand an iterable into more arguments for function calls:

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Operators</h1>

<h2>The ... Operator</h2>

<p>The "Spread" operator can be used to expand an iterable into more arguments for function calls:</p>

<p id="demo"></p>

<script>

const numbers = [23,55,21,87,56];

let maxValue = Math.max(...numbers);

document.getElementById("demo").innerHTML = maxValue;

</script>

</body>

</html>

## The For/Of Loop

The JavaScript for/of statement loops through the values of an iterable objects.

for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

The for/of loop has the following syntax:

for (variable of iterable) {  
  // code block to be executed  
}

variable - For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let, or var.

iterable - An object that has iterable properties.

### Looping over an Array

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For Of Loop</h2>

<p>The for of statement loops through the values of any iterable object:</p>

<p id="demo"></p>

<script>

const cars = ["BMW", "Volvo", "Mini"];

let text = "";

for (let x of cars) {

text += x + "<br>";

}

document.getElementById("demo").innerHTML = text;

</script>

</body>

</html>

### Looping over a String

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For Of Loop</h2>

<p>The for of statement loops through the values of an iterable object.</p>

<p id="demo"></p>

<script>

let language = "JavaScript";

let text = "";

for (let x of language) {

text += x + "<br>";

}

document.getElementById("demo").innerHTML = text;

</script>

</body>

</html>

## JavaScript Maps

**Being able to use an Object as a key is an important Map feature.**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Map Objects</h2>

<p>Creating a Map from an Array:</p>

<p id="demo"></p>

<script>

// Create a Map

const fruits = new Map([

["apples", 500],

["bananas", 300],

["oranges", 200]

]);

document.getElementById("demo").innerHTML = fruits.get("apples");

</script>

</body>

</html>

## JavaScript Sets

Set objects are collections of values. A value in the Set **may only occur once**; it is unique in the Set's collection. You can iterate through the elements of a set in insertion order. The insertion order corresponds to the order in which each element was inserted into the set by the [add()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/add) method successfully (that is, there wasn't an identical element already in the set when add() was called).

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Sets</h2>

<p>Add values to a Set:</p>

<p id="demo"></p>

<script>

// Create a Set

const letters = new Set();

// Add Values to the Set

letters.add("a");

letters.add("b");

letters.add("c");

// Display set.size

document.getElementById("demo").innerHTML = letters.size;

</script>

</body>

</html>

# JavaScript - Modules

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.

For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.

But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

To name some (for historical reasons):

* [AMD](https://en.wikipedia.org/wiki/Asynchronous_module_definition) – one of the most ancient module systems, initially implemented by the library [require.js](http://requirejs.org/).
* [CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1) – the module system created for Node.js server.
* [UMD](https://github.com/umdjs/umd) – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

Now these all slowly became a part of history, but we still can find them in old scripts.

The language-level module system appeared in the standard in 2015, gradually evolved since then, and is now supported by all major browsers and in Node.js. So we’ll study the modern JavaScript modules from now on.

## [What is a module?](https://javascript.info/modules-intro" \l "what-is-a-module)

A module is just a file. One script is one module. As simple as that.

Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

* export keyword labels variables and functions that should be accessible from outside the current module.
* import allows the import of functionality from other modules.

For instance, if we have a file sayHi.js exporting a function:

// 📁 sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

…Then another file may import and use it:

// 📁 main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

The import directive loads the module by path ./sayHi.js relative to the current file, and assigns exported function sayHi to the corresponding variable.

Let’s run the example in-browser.

As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute <script type="module">.

Like this:

**say.js:**

export function sayHi(user) {

return `Hello, ${user}!`;

}

**index.html:**

<!doctype html>

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

The browser automatically fetches and evaluates the imported module (and its imports if needed), and then runs the script.

**Modules work only via HTTP(s), not locally**

If you try to open a web-page locally, via file:// protocol, you’ll find that import/export directives don’t work. Use a local web-server, such as [static-server](https://www.npmjs.com/package/static-server#getting-started) or use the “live server” capability of your editor, such as VS Code [Live Server Extension](https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer) to test modules.

## [Core module features](https://javascript.info/modules-intro" \l "core-module-features)

What’s different in modules, compared to “regular” scripts?

There are core features, valid both for browser and server-side JavaScript.

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

Modules always work in strict mode. E.g. assigning to an undeclared variable will give an error.

<script type="module">

a = 5; // error

</script>

### [Module-level scope](https://javascript.info/modules-intro" \l "module-level-scope)

Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.

In the example below, two scripts are imported, and hello.js tries to use user variable declared in user.js. It fails, because it’s a separate module (you’ll see the error in the console):

**hello.js:**

alert(user); // no such variable (each module has independent variables)

**user.js:**

let user = "John";

**index.html:**

<!doctype html>

<script type="module" src="user.js"></script>

<script type="module" src="hello.js"></script>

Modules should export what they want to be accessible from outside and import what they need.

* user.js should export the user variable.
* hello.js should import it from user.js module.

In other words, with modules we use import/export instead of relying on global variables.

This is the correct variant:

**hello.js:**

import {user} from './user.js';

document.body.innerHTML = user; // John

**user.js:**

export let user = "John";

**index.html:**

<!doctype html>

<script type="module" src="hello.js"></script>

In the browser, if we talk about HTML pages, independent top-level scope also exists for each <script type="module">.

Here are two scripts on the same page, both type="module". They don’t see each other’s top-level variables:

<script type="module">

// The variable is only visible in this module script

let user = "John";

</script>

<script type="module">

alert(user); // Error: user is not defined

</script>

**Please note:**

In the browser, we can make a variable window-level global by explicitly assigning it to a window property, e.g. window.user = "John".

Then all scripts will see it, both with type="module" and without it.

That said, making such global variables is frowned upon. Please try to avoid them.

### [A module code is evaluated only the first time when imported](https://javascript.info/modules-intro" \l "a-module-code-is-evaluated-only-the-first-time-when-imported)

If the same module is imported into multiple other modules, its code is executed only once, upon the first import. Then its exports are given to all further importers.

The one-time evaluation has important consequences, that we should be aware of.

Let’s see a couple of examples.

First, if executing a module code brings side-effects, like showing a message, then importing it multiple times will trigger it only once – the first time:

// 📁 alert.js

alert("Module is evaluated!");

// Import the same module from different files

// 📁 1.js

import `./alert.js`; // Module is evaluated!

// 📁 2.js

import `./alert.js`; // (shows nothing)

The second import shows nothing, because the module has already been evaluated.

There’s a rule: top-level module code should be used for initialization, creation of module-specific internal data structures. If we need to make something callable multiple times – we should export it as a function, like we did with sayHi above.

Now, let’s consider a deeper example.

Let’s say, a module exports an object:

// 📁 admin.js

export let admin = {

name: "John"

};

If this module is imported from multiple files, the module is only evaluated the first time, admin object is created, and then passed to all further importers.

All importers get exactly the one and only admin object:

// 📁 1.js

import {admin} from './admin.js';

admin.name = "Pete";

// 📁 2.js

import {admin} from './admin.js';

alert(admin.name); // Pete

// Both 1.js and 2.js reference the same admin object

// Changes made in 1.js are visible in 2.js

As you can see, when 1.js changes the name property in the imported admin, then 2.js can see the new admin.name.

That’s exactly because the module is executed only once. Exports are generated, and then they are shared between importers, so if something changes the admin object, other importers will see that.

**Such behavior is actually very convenient, because it allows us to configure modules.**

In other words, a module can provide a generic functionality that needs a setup. E.g. authentication needs credentials. Then it can export a configuration object expecting the outer code to assign to it.

Here’s the classical pattern:

1. A module exports some means of configuration, e.g. a configuration object.
2. On the first import we initialize it, write to its properties. The top-level application script may do that.
3. Further imports use the module.

For instance, the admin.js module may provide certain functionality (e.g. authentication), but expect the credentials to come into the config object from outside:

// 📁 admin.js

export let config = { };

export function sayHi() {

alert(`Ready to serve, ${config.user}!`);

}

Here, admin.js exports the config object (initially empty, but may have default properties too).

Then in init.js, the first script of our app, we import config from it and set config.user:

// 📁 init.js

import {config} from './admin.js';

config.user = "Pete";

…Now the module admin.js is configured.

Further importers can call it, and it correctly shows the current user:

// 📁 another.js

import {sayHi} from './admin.js';

sayHi(); // Ready to serve, Pete!

### [import.meta](https://javascript.info/modules-intro" \l "import-meta)

The object import.meta contains the information about the current module.

Its content depends on the environment. In the browser, it contains the URL of the script, or a current webpage URL if inside HTML:

<script type="module">

alert(import.meta.url); // script URL

// for an inline script - the URL of the current HTML-page

</script>

### [In a module, “this” is undefined](https://javascript.info/modules-intro" \l "in-a-module-this-is-undefined)

That’s kind of a minor feature, but for completeness we should mention it.

In a module, top-level this is undefined.

Compare it to non-module scripts, where this is a global object:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

## [Browser-specific features](https://javascript.info/modules-intro" \l "browser-specific-features)

There are also several browser-specific differences of scripts with type="module" compared to regular ones.

You may want to skip this section for now if you’re reading for the first time, or if you don’t use JavaScript in a browser.

### [Module scripts are deferred](https://javascript.info/modules-intro" \l "module-scripts-are-deferred)

Module scripts are always deferred, same effect as defer attribute (described in the chapter [Scripts: async, defer](https://javascript.info/script-async-defer)), for both external and inline scripts.

In other words:

* downloading external module scripts <script type="module" src="..."> doesn’t block HTML processing, they load in parallel with other resources.
* module scripts wait until the HTML document is fully ready (even if they are tiny and load faster than HTML), and then run.
* relative order of scripts is maintained: scripts that go first in the document, execute first.

As a side effect, module scripts always “see” the fully loaded HTML-page, including HTML elements below them.

For instance:

<script type="module">

alert(typeof button); // object: the script can 'see' the button below

// as modules are deferred, the script runs after the whole page is loaded

</script>

Compare to regular script below:

<script>

alert(typeof button); // button is undefined, the script can't see elements below

// regular scripts run immediately, before the rest of the page is processed

</script>

<button id="button">Button</button>

Please note: the second script actually runs before the first! So we’ll see undefined first, and then object.

That’s because modules are deferred, so we wait for the document to be processed. The regular script runs immediately, so we see its output first.

When using modules, we should be aware that the HTML page shows up as it loads, and JavaScript modules run after that, so the user may see the page before the JavaScript application is ready. Some functionality may not work yet. We should put “loading indicators”, or otherwise ensure that the visitor won’t be confused by that.

### [Async works on inline scripts](https://javascript.info/modules-intro" \l "async-works-on-inline-scripts)

For non-module scripts, the async attribute only works on external scripts. Async scripts run immediately when ready, independently of other scripts or the HTML document.

For module scripts, it works on inline scripts as well.

For example, the inline script below has async, so it doesn’t wait for anything.

It performs the import (fetches ./analytics.js) and runs when ready, even if the HTML document is not finished yet, or if other scripts are still pending.

That’s good for functionality that doesn’t depend on anything, like counters, ads, document-level event listeners.

<!-- all dependencies are fetched (analytics.js), and the script runs -->

<!-- doesn't wait for the document or other <script> tags -->

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

### [External scripts](https://javascript.info/modules-intro#external-scripts)

External scripts that have type="module" are different in two aspects:

1. External scripts with the same src run only once:

<!-- the script my.js is fetched and executed only once -->

<script type="module" src="my.js"></script>

<script type="module" src="my.js"></script>

1. External scripts that are fetched from another origin (e.g. another site) require [CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) headers, as described in the chapter [Fetch: Cross-Origin Requests](https://javascript.info/fetch-crossorigin). In other words, if a module script is fetched from another origin, the remote server must supply a header Access-Control-Allow-Origin allowing the fetch.

<!-- another-site.com must supply Access-Control-Allow-Origin -->

<!-- otherwise, the script won't execute -->

<script type="module" src="http://another-site.com/their.js"></script>

That ensures better security by default.

### [No “bare” modules allowed](https://javascript.info/modules-intro" \l "no-bare-modules-allowed)

In the browser, import must get either a relative or absolute URL. Modules without any path are called “bare” modules. Such modules are not allowed in import.

For instance, this import is invalid:

import {sayHi} from 'sayHi'; // Error, "bare" module

// the module must have a path, e.g. './sayHi.js' or wherever the module is

Certain environments, like Node.js or bundle tools allow bare modules, without any path, as they have their own ways for finding modules and hooks to fine-tune them. But browsers do not support bare modules yet.

### [Compatibility, “nomodule”](https://javascript.info/modules-intro" \l "compatibility-nomodule)

Old browsers do not understand type="module". Scripts of an unknown type are just ignored. For them, it’s possible to provide a fallback using the nomodule attribute:

<script type="module">

alert("Runs in modern browsers");

</script>

<script nomodule>

alert("Modern browsers know both type=module and nomodule, so skip this")

alert("Old browsers ignore script with unknown type=module, but execute this.");

</script>

## [Build tools](https://javascript.info/modules-intro" \l "build-tools)

In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as [Webpack](https://webpack.js.org/) and deploy to the production server.

One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.

Build tools do the following:

1. Take a “main” module, the one intended to be put in <script type="module"> in HTML.
2. Analyze its dependencies: imports and then imports of imports etc.
3. Build a single file with all modules (or multiple files, that’s tunable), replacing native import calls with bundler functions, so that it works. “Special” module types like HTML/CSS modules are also supported.
4. In the process, other transformations and optimizations may be applied:

* Unreachable code removed.
* Unused exports removed (“tree-shaking”).
* Development-specific statements like console and debugger removed.
* Modern, bleeding-edge JavaScript syntax may be transformed to older one with similar functionality using [Babel](https://babeljs.io/).
* The resulting file is minified (spaces removed, variables replaced with shorter names, etc).

If we use bundle tools, then as scripts are bundled together into a single file (or few files), import/export statements inside those scripts are replaced by special bundler functions. So the resulting “bundled” script does not contain any import/export, it doesn’t require type="module", and we can put it into a regular script:

<!-- Assuming we got bundle.js from a tool like Webpack -->

<script src="bundle.js"></script>

That said, native modules are also usable. So we won’t be using Webpack here: you can configure it later.

## [Summary](https://javascript.info/modules-intro#summary)

To summarize, the core concepts are:

1. A module is a file. To make import/export work, browsers need <script type="module">. Modules have several differences:

* Deferred by default.
* Async works on inline scripts.
* To load external scripts from another origin (domain/protocol/port), CORS headers are needed.
* Duplicate external scripts are ignored.

1. Modules have their own, local top-level scope and interchange functionality via import/export.
2. Modules always use strict.
3. Module code is executed only once. Exports are created once and shared between importers.

When we use modules, each module implements the functionality and exports it. Then we use import to directly import it where it’s needed. The browser loads and evaluates the scripts automatically.

In production, people often use bundlers such as [Webpack](https://webpack.js.org/) to bundle modules together for performance and other reasons.

# JavaScript - DOM

The backbone of an HTML document is tags.

According to the Document Object Model (DOM), every HTML tag is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.

All these objects are accessible using JavaScript, and we can use them to modify the page.

For example, document.body is the object representing the <body> tag.

Running this code will make the <body> red for 3 seconds:

document.body.style.background = 'red'; // make the background red

setTimeout(() => document.body.style.background = '', 3000); // return back

Here we used style.background to change the background color of document.body, but there are many other properties, such as:

* innerHTML – HTML contents of the node.
* offsetWidth – the node width (in pixels)
* …and so on.

Soon we’ll learn more ways to manipulate the DOM, but first we need to know about its structure.

## [An example of the DOM](https://javascript.info/dom-nodes" \l "an-example-of-the-dom)

Let’s start with the following simple document named **elk.html**:

<!DOCTYPE HTML>

<html>

<head>

<title>About elk</title>

</head>

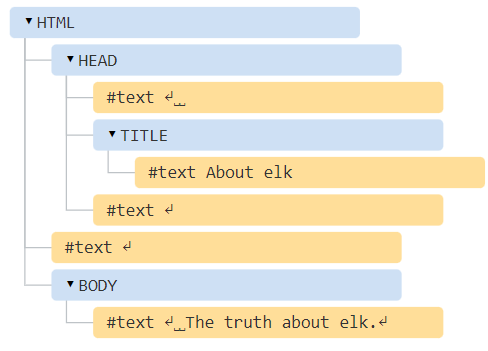
<body>

The truth about elk.

</body>

</html>

The DOM represents HTML as a tree structure of tags. Here’s how it looks:



On the picture above, you can click on element nodes and their children will open/collapse.

Every tree node is an object.

Tags are element nodes (or just elements) and form the tree structure: <html> is at the root, then <head> and <body> are its children, etc.

The text inside elements forms text nodes, labelled as #text. A text node contains only a string. It may not have children and is always a leaf of the tree.

For instance, the <title> tag has the text "About elk".

Please note the special characters in text nodes:

* a newline: ↵ (in JavaScript known as \n)
* a space: ␣

Spaces and newlines are totally valid characters, like letters and digits. They form text nodes and become a part of the DOM. So, for instance, in the example above the <head> tag contains some spaces before <title>, and that text becomes a #text node (it contains a newline and some spaces only).

There are only two top-level exclusions:

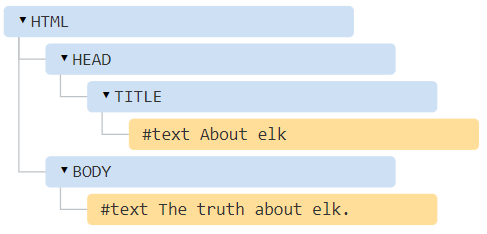
1. Spaces and newlines before <head> are ignored for historical reasons.
2. If we put something after </body>, then that is automatically moved inside the body, at the end, as the HTML spec requires that all content must be inside <body>. So there can’t be any spaces after </body>.

In other cases everything’s straightforward – if there are spaces (just like any character) in the document, then they become text nodes in the DOM, and if we remove them, then there won’t be any.

Here are no space-only text nodes:

<!DOCTYPE HTML>

<html><head><title>About elk</title></head><body>The truth about elk.</body></html>



**Spaces at string start/end and space-only text nodes are usually hidden in tools**

Browser tools (to be covered soon) that work with DOM usually do not show spaces at the start/end of the text and empty text nodes (line-breaks) between tags.

Developer tools save screen space this way.

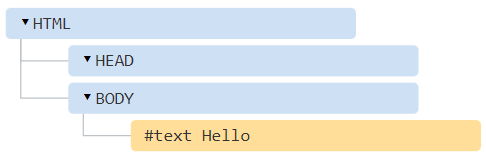
On further DOM pictures we’ll sometimes omit them when they are irrelevant. Such spaces usually do not affect how the document is displayed.

## [Autocorrection](https://javascript.info/dom-nodes" \l "autocorrection)

If the browser encounters malformed HTML, it automatically corrects it when making the DOM.

For instance, the top tag is always <html>. Even if it doesn’t exist in the document, it will exist in the DOM, because the browser will create it. The same goes for <body>.

As an example, if the HTML file is the single word "Hello", the browser will wrap it into <html> and <body>, and add the required <head>, and the DOM will be:



While generating the DOM, browsers automatically process errors in the document, close tags and so on.

A document with unclosed tags:

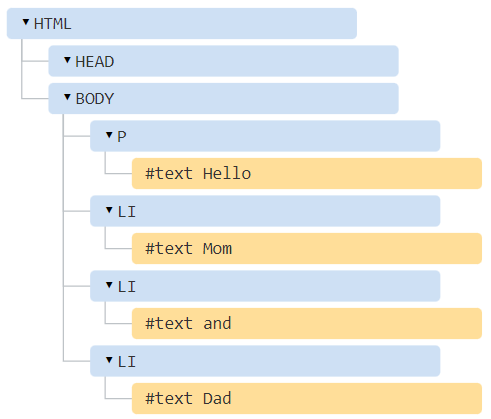
<p>Hello

<li>Mom

<li>and

<li>Dad

…will become a normal DOM as the browser reads tags and restores the missing parts:



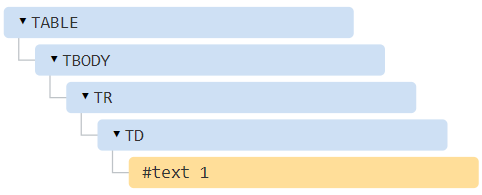
**Tables always have <tbody>**

An interesting “special case” is tables. By DOM specification they must have <tbody> tag, but HTML text may omit it. Then the browser creates <tbody> in the DOM automatically.

For the HTML:

<table id="table"><tr><td>1</td></tr></table>

DOM-structure will be:



You see? The <tbody> appeared out of nowhere. We should keep this in mind while working with tables to avoid surprises.

## [Other node types](https://javascript.info/dom-nodes" \l "other-node-types)

There are some other node types besides elements and text nodes.

For example, comments:

<!DOCTYPE HTML>

<html>

<body>

The truth about elk.

<ol>

<li>An elk is a smart</li>

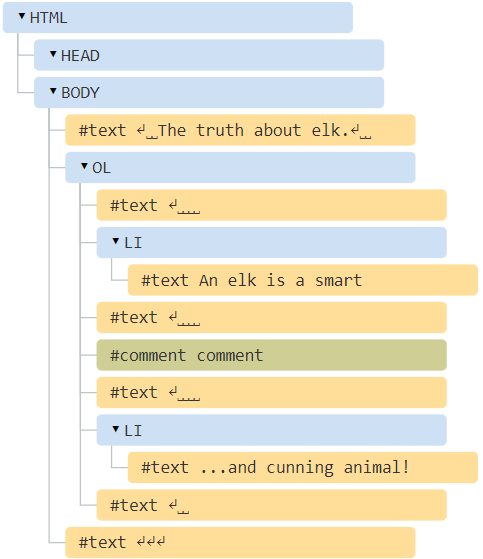
<!-- comment -->

<li>...and cunning animal!</li>

</ol>

</body>

</html>



We can see here a new tree node type – comment node, labeled as #comment, between two text nodes.

We may think – why is a comment added to the DOM? It doesn’t affect the visual representation in any way. But there’s a rule – if something’s in HTML, then it also must be in the DOM tree.

**Everything in HTML, even comments, becomes a part of the DOM.**

Even the <!DOCTYPE...> directive at the very beginning of HTML is also a DOM node. It’s in the DOM tree right before <html>. Few people know about that. We are not going to touch that node, we even don’t draw it on diagrams, but it’s there.

The document object that represents the whole document is, formally, a DOM node as well.

There are [12 node types](https://dom.spec.whatwg.org/#node). In practice we usually work with 4 of them:

1. document – the “entry point” into DOM.
2. element nodes – HTML-tags, the tree building blocks.
3. text nodes – contain text.
4. comments – sometimes we can put information there, it won’t be shown, but JS can read it from the DOM.

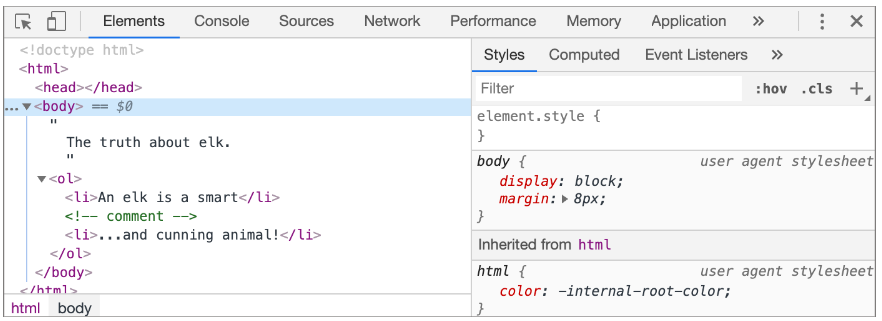
## [See it for yourself](https://javascript.info/dom-nodes" \l "see-it-for-yourself)

To see the DOM structure in real-time, try [Live DOM Viewer](http://software.hixie.ch/utilities/js/live-dom-viewer/). Just type in the document, and it will show up as a DOM at an instant.

Another way to explore the DOM is to use the browser developer tools. Actually, that’s what we use when developing.

To do so, open the web page [elk.html](https://javascript.info/article/dom-nodes/elk.html), turn on the browser developer tools and switch to the Elements tab.

It should look like this:

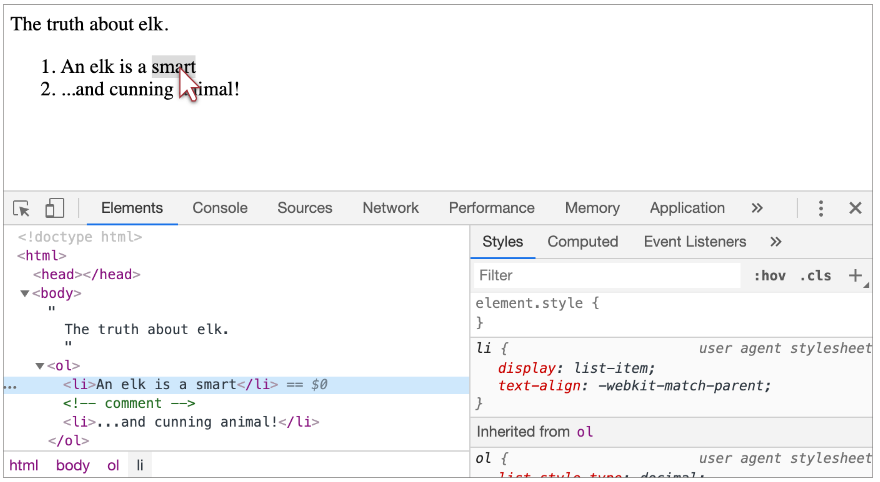


You can see the DOM, click on elements, see their details and so on.

Please note that the DOM structure in developer tools is simplified. Text nodes are shown just as text. And there are no “blank” (space only) text nodes at all. That’s fine, because most of the time we are interested in element nodes.

Clicking the  button in the left-upper corner allows us to choose a node from the webpage using a mouse (or other pointer devices) and “inspect” it (scroll to it in the Elements tab). This works great when we have a huge HTML page (and corresponding huge DOM) and would like to see the place of a particular element in it.

Another way to do it would be just right-clicking on a webpage and selecting “Inspect” in the context menu.



At the right part of the tools there are the following subtabs:

* **Styles** – we can see CSS applied to the current element rule by rule, including built-in rules (gray). Almost everything can be edited in-place, including the dimensions/margins/paddings of the box below.
* **Computed** – to see CSS applied to the element by property: for each property we can see a rule that gives it (including CSS inheritance and such).
* **Event Listeners** – to see event listeners attached to DOM elements (we’ll cover them in the next part of the tutorial).
* …and so on.

The best way to study them is to click around. Most values are editable in-place.

## [Interaction with console](https://javascript.info/dom-nodes" \l "interaction-with-console)

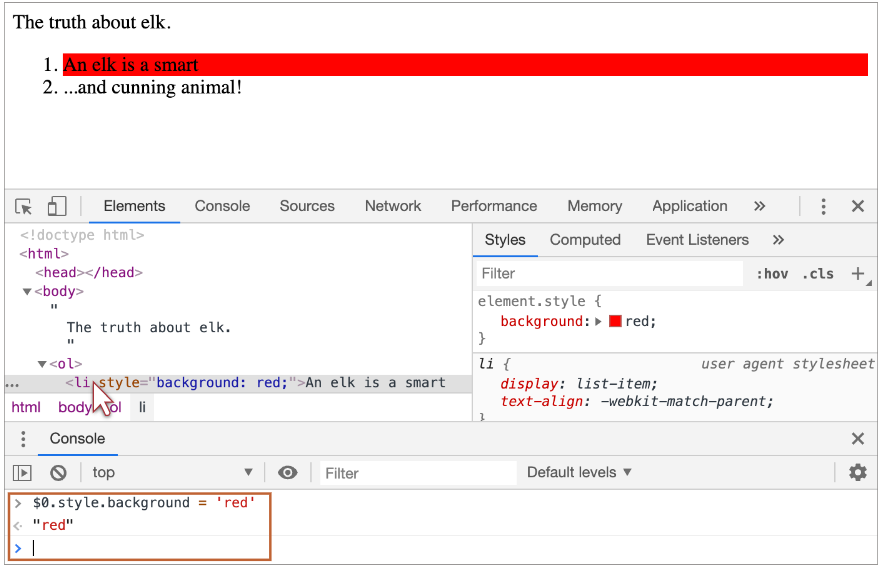
As we work the DOM, we also may want to apply JavaScript to it. Like: get a node and run some code to modify it, to see the result. Here are few tips to travel between the Elements tab and the console.

For the start:

1. Select the first <li> in the Elements tab.
2. Press Esc – it will open console right below the Elements tab.

Now the last selected element is available as $0, the previously selected is $1 etc.

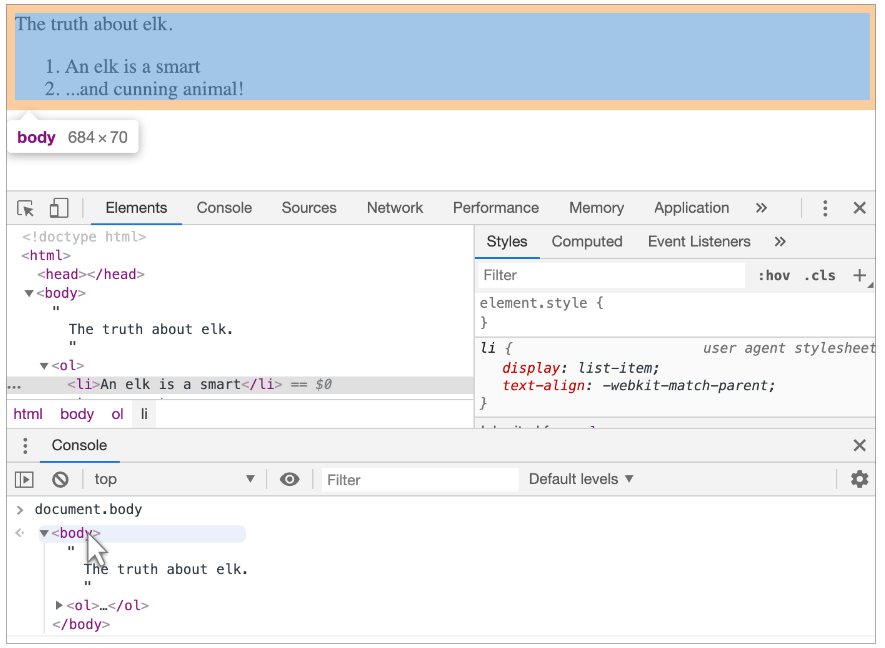
We can run commands on them. For instance, $0.style.background = 'red' makes the selected list item red, like this:



That’s how to get a node from Elements in Console.

There’s also a road back. If there’s a variable referencing a DOM node, then we can use the command inspect(node) in Console to see it in the Elements pane.

Or we can just output the DOM node in the console and explore “in-place”, like document.body below:



That’s for debugging purposes of course. From the next chapter on we’ll access and modify DOM using JavaScript.

The browser developer tools are a great help in development: we can explore the DOM, try things and see what goes wrong.

## [Summary](https://javascript.info/dom-nodes#summary)

An HTML/XML document is represented inside the browser as the DOM tree.

* Tags become element nodes and form the structure.
* Text becomes text nodes.
* …etc, everything in HTML has its place in DOM, even comments.

We can use developer tools to inspect DOM and modify it manually.

Here we covered the basics, the most used and important actions to start with. There’s an extensive documentation about Chrome Developer Tools at <https://developers.google.com/web/tools/chrome-devtools>. The best way to learn the tools is to click here and there, read menus: most options are obvious. Later, when you know them in general, read the docs and pick up the rest.

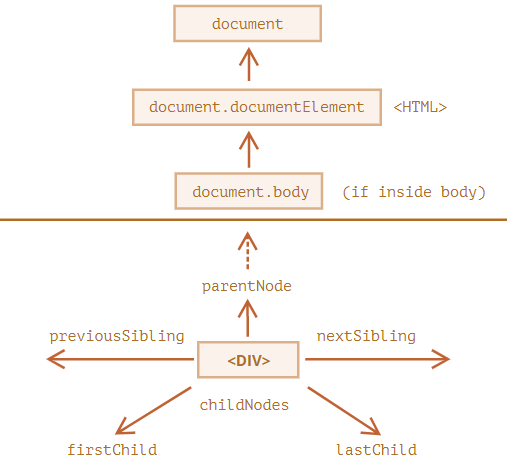
DOM nodes have properties and methods that allow us to travel between them, modify them, move around the page, and more.

# Walking the DOM

The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object.

All operations on the DOM start with the document object. That’s the main “entry point” to DOM. From it we can access any node.

Here’s a picture of links that allow for travel between DOM nodes:



Let’s discuss them in more detail.

## [On top: documentElement and body](https://javascript.info/dom-navigation" \l "on-top-documentelement-and-body)

The topmost tree nodes are available directly as document properties:

<html> = document.documentElement

The topmost document node is document.documentElement. That’s the DOM node of the <html> tag.

<body> = document.body

Another widely used DOM node is the <body> element – document.body.

<head> = document.head

The <head> tag is available as document.head.

**There’s a catch: document.body can be null**

A script cannot access an element that doesn’t exist at the moment of running.

In particular, if a script is inside <head>, then document.body is unavailable, because the browser did not read it yet.

So, in the example below the first alert shows null:

<html>

<head>

<script>

alert( "From HEAD: " + document.body ); // null, there's no <body> yet

</script>

</head>

<body>

<script>

alert( "From BODY: " + document.body ); // HTMLBodyElement, now it exists

</script>

</body>

</html>

**In the DOM world null means “doesn’t exist”**

In the DOM, the null value means “doesn’t exist” or “no such node”.

## [Children: childNodes, firstChild, lastChild](https://javascript.info/dom-navigation" \l "children-childnodes-firstchild-lastchild)

There are two terms that we’ll use from now on:

* **Child nodes (or children)** – elements that are direct children. In other words, they are nested exactly in the given one. For instance, <head> and <body> are children of <html> element.
* **Descendants** – all elements that are nested in the given one, including children, their children and so on.

For instance, here <body> has children <div> and <ul> (and few blank text nodes):

<html>

<body>

<div>Begin</div>

<ul>

<li>

<b>Information</b>

</li>

</ul>

</body>

</html>

…And descendants of <body> are not only direct children <div>, <ul> but also more deeply nested elements, such as <li> (a child of <ul>) and <b> (a child of <li>) – the entire subtree.

**The childNodes collection lists all child nodes, including text nodes.**

The example below shows children of document.body:

<html>

<body>

<div>Begin</div>

<ul>

<li>Information</li>

</ul>

<div>End</div>

<script>

for (let i = 0; i < document.body.childNodes.length; i++) {

alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT

}

</script>

...more stuff...

</body>

</html>

Please note an interesting detail here. If we run the example above, the last element shown is <script>. In fact, the document has more stuff below, but at the moment of the script execution the browser did not read it yet, so the script doesn’t see it.

**Properties firstChild and lastChild give fast access to the first and last children.**

They are just shorthands. If there exist child nodes, then the following is always true:

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

There’s also a special function elem.hasChildNodes() to check whether there are any child nodes.

### [DOM collections](https://javascript.info/dom-navigation" \l "dom-collections)

As we can see, childNodes looks like an array. But actually it’s not an array, but rather a collection – a special array-like iterable object.

There are two important consequences:

1. We can use for..of to iterate over it:

for (let node of document.body.childNodes) {

alert(node); // shows all nodes from the collection

}

That’s because it’s iterable (provides the Symbol.iterator property, as required).

1. Array methods won’t work, because it’s not an array:

alert(document.body.childNodes.filter); // undefined (there's no filter method!)

The first thing is nice. The second is tolerable, because we can use Array.from to create a “real” array from the collection, if we want array methods:

alert( Array.from(document.body.childNodes).filter ); // function

**DOM collections are read-only**

DOM collections, and even more – all navigation properties listed in this chapter are read-only.

We can’t replace a child by something else by assigning childNodes[i] = ....

Changing DOM needs other methods. We will see them in the next chapter.

**DOM collections are live**

Almost all DOM collections with minor exceptions are live. In other words, they reflect the current state of DOM.

If we keep a reference to elem.childNodes, and add/remove nodes into DOM, then they appear in the collection automatically.

**Don’t use for..in to loop over collections**

Collections are iterable using for..of. Sometimes people try to use for..in for that.

Please, don’t. The for..in loop iterates over all enumerable properties. And collections have some “extra” rarely used properties that we usually do not want to get:

<body>

<script>

// shows 0, 1, length, item, values and more.

for (let prop in document.body.childNodes) alert(prop);

</script>

</body>

## [Siblings and the parent](https://javascript.info/dom-navigation" \l "siblings-and-the-parent)

Siblings are nodes that are children of the same parent.

For instance, here <head> and <body> are siblings:

<html>

<head>...</head><body>...</body>

</html>

* <body> is said to be the “next” or “right” sibling of <head>,
* <head> is said to be the “previous” or “left” sibling of <body>.

The next sibling is in nextSibling property, and the previous one – in previousSibling.

The parent is available as parentNode.

For example:

// parent of <body> is <html>

alert( document.body.parentNode === document.documentElement ); // true

// after <head> goes <body>

alert( document.head.nextSibling ); // HTMLBodyElement

// before <body> goes <head>

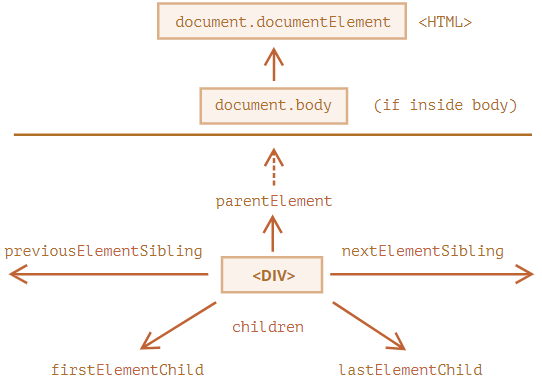
alert( document.body.previousSibling ); // HTMLHeadElement

## [Element-only navigation](https://javascript.info/dom-navigation" \l "element-only-navigation)

Navigation properties listed above refer to all nodes. For instance, in childNodes we can see both text nodes, element nodes, and even comment nodes if they exist.

But for many tasks we don’t want text or comment nodes. We want to manipulate element nodes that represent tags and form the structure of the page.

So let’s see more navigation links that only take element nodes into account:



The links are similar to those given above, just with Element word inside:

* children – only those children that are element nodes.
* firstElementChild, lastElementChild – first and last element children.
* previousElementSibling, nextElementSibling – neighbor elements.
* parentElement – parent element.

**Why parentElement? Can the parent be not an element?**

The parentElement property returns the “element” parent, while parentNode returns “any node” parent. These properties are usually the same: they both get the parent.

With the one exception of document.documentElement:

alert( document.documentElement.parentNode ); // document

alert( document.documentElement.parentElement ); // null

The reason is that the root node document.documentElement (<html>) has document as its parent. But document is not an element node, so parentNode returns it and parentElement does not.

This detail may be useful when we want to travel up from an arbitrary element elem to <html>, but not to the document:

while(elem = elem.parentElement) { // go up till <html>

alert( elem );

}

Let’s modify one of the examples above: replace childNodes with children. Now it shows only elements:

<html>

<body>

<div>Begin</div>

<ul>

<li>Information</li>

</ul>

<div>End</div>

<script>

for (let elem of document.body.children) {

alert(elem); // DIV, UL, DIV, SCRIPT

}

</script>

...

</body>

</html>

## [More links: tables](https://javascript.info/dom-navigation" \l "dom-navigation-tables)

Till now we described the basic navigation properties.

Certain types of DOM elements may provide additional properties, specific to their type, for convenience.

Tables are a great example of that, and represent a particularly important case:

**The <table>** element supports (in addition to the given above) these properties:

* table.rows – the collection of <tr> elements of the table.
* table.caption/tHead/tFoot – references to elements <caption>, <thead>, <tfoot>.
* table.tBodies – the collection of <tbody> elements (can be many according to the standard, but there will always be at least one – even if it is not in the source HTML, the browser will put it in the DOM).

**<thead>, <tfoot>, <tbody>** elements provide the rows property:

* tbody.rows – the collection of <tr> inside.

**<tr>:**

* tr.cells – the collection of <td> and <th> cells inside the given <tr>.
* tr.sectionRowIndex – the position (index) of the given <tr> inside the enclosing <thead>/<tbody>/<tfoot>.
* tr.rowIndex – the number of the <tr> in the table as a whole (including all table rows).

**<td> and <th>:**

* td.cellIndex – the number of the cell inside the enclosing <tr>.

An example of usage:

<table id="table">

<tr>

<td>one</td><td>two</td>

</tr>

<tr>

<td>three</td><td>four</td>

</tr>

</table>

<script>

// get td with "two" (first row, second column)

let td = table.rows[0].cells[1];

td.style.backgroundColor = "red"; // highlight it

</script>

The specification: [tabular data](https://html.spec.whatwg.org/multipage/tables.html).

There are also additional navigation properties for HTML forms. We’ll look at them later when we start working with forms.

## [Summary](https://javascript.info/dom-navigation#summary)

Given a DOM node, we can go to its immediate neighbors using navigation properties.

There are two main sets of them:

* For all nodes: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.
* For element nodes only: parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.

Some types of DOM elements, e.g. tables, provide additional properties and collections to access their content.

## [Tasks](https://javascript.info/dom-navigation#tasks)

### [DOM children](https://javascript.info/dom-navigation" \l "dom-children)

importance: 5

Look at this page:

<html>

<body>

<div>Users:</div>

<ul>

<li>John</li>

<li>Pete</li>

</ul>

</body>

</html>

For each of the following, give at least one way of how to access them:

* The <div> DOM node?
* The <ul> DOM node?
* The second <li> (with Pete)?

**Solution**

There are many ways, for instance:

The <div> DOM node:

document.body.firstElementChild

// or

document.body.children[0]

// or (the first node is space, so we take 2nd)

document.body.childNodes[1]

The <ul> DOM node:

document.body.lastElementChild

// or

document.body.children[1]

The second <li> (with Pete):

// get <ul>, and then get its last element child

document.body.lastElementChild.lastElementChild

### [The sibling question](https://javascript.info/dom-navigation" \l "the-sibling-question)

importance: 5

If elem – is an arbitrary DOM element node…

* Is it true that elem.lastChild.nextSibling is always null?
* Is it true that elem.children[0].previousSibling is always null ?

**Solution**

1. Yes, true. The element elem.lastChild is always the last one, it has no nextSibling.
2. No, wrong, because elem.children[0] is the first child *among elements*. But there may exist non-element nodes before it. So previousSibling may be a text node.

Please note: for both cases if there are no children, then there will be an error.

If there are no children, elem.lastChild is null, so we can’t access elem.lastChild.nextSibling. And the collection elem.children is empty (like an empty array []).

### [Select all diagonal cells](https://javascript.info/dom-navigation" \l "select-all-diagonal-cells)

importance: 5

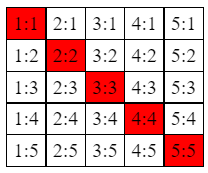
Write the code to paint all diagonal table cells in red.

You’ll need to get all diagonal <td> from the <table> and paint them using the code:

// td should be the reference to the table cell

td.style.backgroundColor = 'red';

The result should be:



**Solution**

We’ll be using rows and cells properties to access diagonal table cells.

<!DOCTYPE HTML>

<html>

<head>

  <style>

    table {

      border-collapse: collapse;

    }

    td {

      border: 1px solid black;

      padding: 3px 5px;

    }

  </style>

</head>

<body>

  <table>

    <tr>

      <td>1:1</td>

      <td>2:1</td>

      <td>3:1</td>

      <td>4:1</td>

      <td>5:1</td>

    </tr>

    <tr>

      <td>1:2</td>

      <td>2:2</td>

      <td>3:2</td>

      <td>4:2</td>

      <td>5:2</td>

    </tr>

    <tr>

      <td>1:3</td>

      <td>2:3</td>

      <td>3:3</td>

      <td>4:3</td>

      <td>5:3</td>

    </tr>

    <tr>

      <td>1:4</td>

      <td>2:4</td>

      <td>3:4</td>

      <td>4:4</td>

      <td>5:4</td>

    </tr>

    <tr>

      <td>1:5</td>

      <td>2:5</td>

      <td>3:5</td>

      <td>4:5</td>

      <td>5:5</td>

    </tr>

  </table>

  <script>

    let table = document.body.firstElementChild;

    for (let i = 0; i < table.rows.length; i++) {

      let row = table.rows[i];

      row.cells[i].style.backgroundColor = 'red';

    }

  </script>

</body>

</html>

# Searching: getElement\*, querySelector\*

DOM navigation properties are great when elements are close to each other. What if they are not? How to get an arbitrary element of the page?

There are additional searching methods for that.

## [document.getElementById or just id](https://javascript.info/searching-elements-dom" \l "document-getelementbyid-or-just-id)

If an element has the id attribute, we can get the element using the method document.getElementById(id), no matter where it is.

For instance:

<div id="elem">

<div id="elem-content">Element</div>

</div>

<script>

// get the element

let elem = document.getElementById('elem');

// make its background red

elem.style.background = 'red';

</script>

Also, there’s a global variable named by id that references the element:

<div id="elem">

<div id="elem-content">Element</div>

</div>

<script>

// elem is a reference to DOM-element with id="elem"

elem.style.background = 'red';

// id="elem-content" has a hyphen inside, so it can't be a variable name

// ...but we can access it using square brackets: window['elem-content']

</script>

…That’s unless we declare a JavaScript variable with the same name, then it takes precedence:

<div id="elem"></div>

<script>

let elem = 5; // now elem is 5, not a reference to <div id="elem">

alert(elem); // 5

</script>

**Please don’t use id-named global variables to access elements**

This behavior is described [in the specification](http://www.whatwg.org/specs/web-apps/current-work/#dom-window-nameditem), so it’s a kind of standard. But it is supported mainly for compatibility.

The browser tries to help us by mixing namespaces of JS and DOM. That’s fine for simple scripts, inlined into HTML, but generally isn’t a good thing. There may be naming conflicts. Also, when one reads JS code and doesn’t have HTML in view, it’s not obvious where the variable comes from.

Here in the tutorial we use id to directly reference an element for brevity, when it’s obvious where the element comes from.

In real life document.getElementById is the preferred method.

**The id must be unique**

The id must be unique. There can be only one element in the document with the given id.

If there are multiple elements with the same id, then the behavior of methods that use it is unpredictable, e.g. document.getElementById may return any of such elements at random. So please stick to the rule and keep id unique.

**Only document.getElementById, not anyElem.getElementById**

The method getElementById can be called only on document object. It looks for the given id in the whole document.

## [querySelectorAll](https://javascript.info/searching-elements-dom" \l "querySelectorAll)

By far, the most versatile method, elem.querySelectorAll(css) returns all elements inside elem matching the given CSS selector.

Here we look for all <li> elements that are last children:

<ul>

<li>The</li>

<li>test</li>

</ul>

<ul>

<li>has</li>

<li>passed</li>

</ul>

<script>

let elements = document.querySelectorAll('ul > li:last-child');

for (let elem of elements) {

alert(elem.innerHTML); // "test", "passed"

}

</script>

This method is indeed powerful, because any CSS selector can be used.

**Can use pseudo-classes as well**

Pseudo-classes in the CSS selector like :hover and :active are also supported. For instance, document.querySelectorAll(':hover') will return the collection with elements that the pointer is over now (in nesting order: from the outermost <html> to the most nested one).

## [querySelector](https://javascript.info/searching-elements-dom" \l "querySelector)

The call to elem.querySelector(css) returns the first element for the given CSS selector.

In other words, the result is the same as elem.querySelectorAll(css)[0], but the latter is looking for all elements and picking one, while elem.querySelector just looks for one. So it’s faster and also shorter to write.

## [matches](https://javascript.info/searching-elements-dom" \l "matches)

Previous methods were searching the DOM.

The [elem.matches(css)](https://dom.spec.whatwg.org/#dom-element-matches) does not look for anything, it merely checks if elem matches the given CSS-selector. It returns true or false.

The method comes in handy when we are iterating over elements (like in an array or something) and trying to filter out those that interest us.

For instance:

<a href="http://example.com/file.zip">...</a>

<a href="http://ya.ru">...</a>

<script>

// can be any collection instead of document.body.children

for (let elem of document.body.children) {

if (elem.matches('a[href$="zip"]')) {

alert("The archive reference: " + elem.href );

}

}

</script>

## [closest](https://javascript.info/searching-elements-dom" \l "closest)

Ancestors of an element are: parent, the parent of parent, its parent and so on. The ancestors together form the chain of parents from the element to the top.

The method elem.closest(css) looks for the nearest ancestor that matches the CSS-selector. The elem itself is also included in the search.

In other words, the method closest goes up from the element and checks each of parents. If it matches the selector, then the search stops, and the ancestor is returned.

For instance:

<h1>Contents</h1>

<div class="contents">

<ul class="book">

<li class="chapter">Chapter 1</li>

<li class="chapter">Chapter 2</li>

</ul>

</div>

<script>

let chapter = document.querySelector('.chapter'); // LI

alert(chapter.closest('.book')); // UL

alert(chapter.closest('.contents')); // DIV

alert(chapter.closest('h1')); // null (because h1 is not an ancestor)

</script>

## [getElementsBy\*](https://javascript.info/searching-elements-dom" \l "getelementsby)

There are also other methods to look for nodes by a tag, class, etc.

Today, they are mostly history, as querySelector is more powerful and shorter to write.

So here we cover them mainly for completeness, while you can still find them in the old scripts.

* elem.getElementsByTagName(tag) looks for elements with the given tag and returns the collection of them. The tag parameter can also be a star "\*" for “any tags”.
* elem.getElementsByClassName(className) returns elements that have the given CSS class.
* document.getElementsByName(name) returns elements with the given name attribute, document-wide. Very rarely used.

For instance:

// get all divs in the document

let divs = document.getElementsByTagName('div');

Let’s find all input tags inside the table:

<table id="table">

<tr>

<td>Your age:</td>

<td>

<label>

<input type="radio" name="age" value="young" checked> less than 18

</label>

<label>

<input type="radio" name="age" value="mature"> from 18 to 50

</label>

<label>

<input type="radio" name="age" value="senior"> more than 60

</label>

</td>

</tr>

</table>

<script>

let inputs = table.getElementsByTagName('input');

for (let input of inputs) {

alert( input.value + ': ' + input.checked );

}

</script>

**Don’t forget the "s" letter!**

Novice developers sometimes forget the letter "s". That is, they try to call getElementByTagName instead of getElement**s**ByTagName.

The "s" letter is absent in getElementById, because it returns a single element. But getElementsByTagName returns a collection of elements, so there’s "s" inside.

**It returns a collection, not an element!**

Another widespread novice mistake is to write:

// doesn't work

document.getElementsByTagName('input').value = 5;

That won’t work, because it takes a collection of inputs and assigns the value to it rather than to elements inside it.

We should either iterate over the collection or get an element by its index, and then assign, like this:

// should work (if there's an input)

document.getElementsByTagName('input')[0].value = 5;

Looking for .article elements:

<form name="my-form">

<div class="article">Article</div>

<div class="long article">Long article</div>

</form>

<script>

// find by name attribute

let form = document.getElementsByName('my-form')[0];

// find by class inside the form

let articles = form.getElementsByClassName('article');

alert(articles.length); // 2, found two elements with class "article"

</script>

## [Live collections](https://javascript.info/searching-elements-dom" \l "live-collections)

All methods "getElementsBy\*" return a live collection. Such collections always reflect the current state of the document and “auto-update” when it changes.

In the example below, there are two scripts.

1. The first one creates a reference to the collection of <div>. As of now, its length is 1.
2. The second scripts runs after the browser meets one more <div>, so its length is 2.

<div>First div</div>

<script>

let divs = document.getElementsByTagName('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 2

</script>

In contrast, querySelectorAll returns a static collection. It’s like a fixed array of elements.

If we use it instead, then both scripts output 1:

<div>First div</div>

<script>

let divs = document.querySelectorAll('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 1

</script>

Now we can easily see the difference. The static collection did not increase after the appearance of a new div in the document.

## [Summary](https://javascript.info/searching-elements-dom#summary)

There are 6 main methods to search for nodes in DOM:

| **Method** | **Searches by...** | **Can call on an element?** | **Live?** |
| --- | --- | --- | --- |
| querySelector | CSS-selector | ✔ | - |
| querySelectorAll | CSS-selector | ✔ | - |
| getElementById | id | - | - |
| getElementsByName | name | - | ✔ |
| getElementsByTagName | tag or '\*' | ✔ | ✔ |
| getElementsByClassName | class | ✔ | ✔ |

By far the most used are querySelector and querySelectorAll, but getElement(s)By\* can be sporadically helpful or found in the old scripts.

Besides that:

* There is elem.matches(css) to check if elem matches the given CSS selector.
* There is elem.closest(css) to look for the nearest ancestor that matches the given CSS-selector. The elem itself is also checked.

And let’s mention one more method here to check for the child-parent relationship, as it’s sometimes useful:

* elemA.contains(elemB) returns true if elemB is inside elemA (a descendant of elemA) or when elemA==elemB.

## [Tasks](https://javascript.info/searching-elements-dom#tasks)

### [Search for elements](https://javascript.info/searching-elements-dom" \l "search-for-elements)

importance: 4

Here’s the document with the table and form.

How to find?…

1. The table with id="age-table".
2. All label elements inside that table (there should be 3 of them).
3. The first td in that table (with the word “Age”).
4. The form with name="search".
5. The first input in that form.
6. The last input in that form.

Open the page [table.html](https://javascript.info/task/find-elements/table.html) in a separate window and make use of browser tools for that.

**table.html:**

<!DOCTYPE HTML>

<html>

<body>

<form name="search">

<label>Search the site:

<input type="text" name="search">

</label>

<input type="submit" value="Search!">

</form>

<hr>

<form name="search-person">

Search the visitors:

<table id="age-table">

<tr>

<td>Age:</td>

<td id="age-list">

<label>

<input type="radio" name="age" value="young">less than 18</label>

<label>

<input type="radio" name="age" value="mature">18-50</label>

<label>

<input type="radio" name="age" value="senior">more than 50</label>

</td>

</tr>

<tr>

<td>Additionally:</td>

<td>

<input type="text" name="info[0]">

<input type="text" name="info[1]">

<input type="text" name="info[2]">

</td>

</tr>

</table>

<input type="submit" value="Search!">

</form>

</body>

</html>

**Solution**

There are many ways to do it.

Here are some of them:

// 1. The table with `id="age-table"`.

let table = document.getElementById('age-table')

// 2. All label elements inside that table

table.getElementsByTagName('label')

// or

document.querySelectorAll('#age-table label')

// 3. The first td in that table (with the word "Age")

table.rows[0].cells[0]

// or

table.getElementsByTagName('td')[0]

// or

table.querySelector('td')

// 4. The form with the name "search"

// assuming there's only one element with name="search" in the document

let form = document.getElementsByName('search')[0]

// or, form specifically

document.querySelector('form[name="search"]')

// 5. The first input in that form.

form.getElementsByTagName('input')[0]

// or

form.querySelector('input')

// 6. The last input in that form

let inputs = form.querySelectorAll('input') // find all inputs

inputs[inputs.length-1] // take the last one

# JavaScript – Events

<https://javascript.info/introduction-browser-events>

An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

Here’s a list of the most useful DOM events, just to take a look at:

**Mouse events:**

* click – when the mouse clicks on an element (touchscreen devices generate it on a tap).
* contextmenu – when the mouse right-clicks on an element.
* mouseover / mouseout – when the mouse cursor comes over / leaves an element.
* mousedown / mouseup – when the mouse button is pressed / released over an element.
* mousemove – when the mouse is moved.

**Keyboard events:**

* keydown and keyup – when a keyboard key is pressed and released.

**Form element events:**

* submit – when the visitor submits a <form>.
* focus – when the visitor focuses on an element, e.g. on an <input>.

**Document events:**

* DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

**CSS events:**

* transitionend – when a CSS-animation finishes.

There are many other events

## [Event handlers](https://javascript.info/introduction-browser-events" \l "event-handlers)

To react on events we can assign a handler – a function that runs in case of an event.

Handlers are a way to run JavaScript code in case of user actions.

There are several ways to assign a handler. Let’s see them, starting from the simplest one.

### [HTML-attribute](https://javascript.info/introduction-browser-events" \l "html-attribute)

A handler can be set in HTML with an attribute named on<event>.

For instance, to assign a click handler for an input, we can use onclick, like here:

<input value="Click me" onclick="alert('Click!')" type="button">

On mouse click, the code inside onclick runs.

Please note that inside onclick we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: onclick="alert("Click!")", then it won’t work right.

An HTML-attribute is not a convenient place to write a lot of code, so we’d better create a JavaScript function and call it there.

Here a click runs the function countRabbits():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Rabbit number " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">

As we know, HTML attribute names are not case-sensitive, so ONCLICK works as well as onClick and onCLICK… But usually attributes are lowercased: onclick.

### [DOM property](https://javascript.info/introduction-browser-events" \l "dom-property)

We can assign a handler using a DOM property on<event>.

For instance, elem.onclick:

<input id="elem" type="button" value="Click me">

<script>

elem.onclick = function() {

alert('Thank you');

};

</script>

If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property.

So this way is actually the same as the previous one.

These two code pieces work the same:

1. Only HTML:

<input type="button" onclick="alert('Click!')" value="Button">

1. HTML + JS:

<input type="button" id="button" value="Button">

<script>

button.onclick = function() {

alert('Click!');

};

</script>

In the first example, the HTML attribute is used to initialize the button.onclick, while in the second example – the script, that’s all the difference.

**As there’s only one onclick property, we can’t assign more than one event handler.**

In the example below adding a handler with JavaScript overwrites the existing handler:

<input type="button" id="elem" onclick="alert('Before')" value="Click me">

<script>

elem.onclick = function() { // overwrites the existing handler

alert('After'); // only this will be shown

};

</script>

To remove a handler – assign elem.onclick = null.

## [Accessing the element: this](https://javascript.info/introduction-browser-events" \l "accessing-the-element-this)

The value of this inside a handler is the element. The one which has the handler on it.

In the code below button shows its contents using this.innerHTML:

<button onclick="alert(this.innerHTML)">Click me</button>

## [Possible mistakes](https://javascript.info/introduction-browser-events" \l "possible-mistakes)

If you’re starting to work with events – please note some subtleties.

We can set an existing function as a handler:

function sayThanks() {

alert('Thanks!');

}

elem.onclick = sayThanks;

But be careful: the function should be assigned as sayThanks, not sayThanks().

// right

button.onclick = sayThanks;

// wrong

button.onclick = sayThanks();

If we add parentheses, then sayThanks() becomes a function call. So the last line actually takes the result of the function execution, that is undefined (as the function returns nothing), and assigns it to onclick. That doesn’t work.

…On the other hand, in the markup we do need the parentheses:

<input type="button" id="button" onclick="sayThanks()">

The difference is easy to explain. When the browser reads the attribute, it creates a handler function with body from the attribute content.

So the markup generates this property:

button.onclick = function() {

sayThanks(); // <-- the attribute content goes here

};

**Don’t use setAttribute for handlers.**

Such a call won’t work:

// a click on <body> will generate errors,

// because attributes are always strings, function becomes a string

document.body.setAttribute('onclick', function() { alert(1) });

**DOM-property case matters.**

Assign a handler to elem.onclick, not elem.ONCLICK, because DOM properties are case-sensitive.

## [addEventListener](https://javascript.info/introduction-browser-events" \l "addeventlistener)

The fundamental problem of the aforementioned ways to assign handlers is that we can’t assign multiple handlers to one event.

Let’s say, one part of our code wants to highlight a button on click, and another one wants to show a message on the same click.

We’d like to assign two event handlers for that. But a new DOM property will overwrite the existing one:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); } // replaces the previous handler

Developers of web standards understood that long ago and suggested an alternative way of managing handlers using the special methods addEventListener and removeEventListener which aren’t bound by such constraint.

The syntax to add a handler:

element.addEventListener(event, handler, [options]);

**event**

Event name, e.g. "click".

**handler**

The handler function.

**options**

An additional optional object with properties:

* once: if true, then the listener is automatically removed after it triggers.
* capture: the phase where to handle the event, to be covered later in the chapter [Bubbling and capturing](https://javascript.info/bubbling-and-capturing). For historical reasons, options can also be false/true, that’s the same as {capture: false/true}.
* passive: if true, then the handler will not call preventDefault(), we’ll explain that later in [Browser default actions](https://javascript.info/default-browser-action).

To remove the handler, use removeEventListener:

element.removeEventListener(event, handler, [options]);

**Removal requires the same function**

To remove a handler we should pass exactly the same function as was assigned.

This doesn’t work:

elem.addEventListener( "click" , () => alert('Thanks!'));

// ....

elem.removeEventListener( "click", () => alert('Thanks!'));

The handler won’t be removed, because removeEventListener gets another function – with the same code, but that doesn’t matter, as it’s a different function object.

Here’s the right way:

function handler() {

alert( 'Thanks!' );

}

input.addEventListener("click", handler);

// ....

input.removeEventListener("click", handler);

Please note – if we don’t store the function in a variable, then we can’t remove it. There’s no way to “read back” handlers assigned by addEventListener.

Multiple calls to addEventListener allow it to add multiple handlers, like this:

<input id="elem" type="button" value="Click me"/>

<script>

function handler1() {

alert('Thanks!');

};

function handler2() {

alert('Thanks again!');

}

elem.onclick = () => alert("Hello");

elem.addEventListener("click", handler1); // Thanks!

elem.addEventListener("click", handler2); // Thanks again!

</script>

As we can see in the example above, we can set handlers both using a DOM-property and addEventListener. But generally we use only one of these ways.

**For some events, handlers only work with addEventListener**

There exist events that can’t be assigned via a DOM-property. Only with addEventListener.

For instance, the DOMContentLoaded event, that triggers when the document is loaded and the DOM has been built.

// will never run

document.onDOMContentLoaded = function() {

alert("DOM built");

};

// this way it works

document.addEventListener("DOMContentLoaded", function() {

alert("DOM built");

});

So addEventListener is more universal. Although, such events are an exception rather than the rule.

## [Event object](https://javascript.info/introduction-browser-events" \l "event-object)

To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keydown”, but what were the pointer coordinates? Which key was pressed? And so on.

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.

Here’s an example of getting pointer coordinates from the event object:

<input type="button" value="Click me" id="elem">

<script>

elem.onclick = function(event) {

// show event type, element and coordinates of the click

alert(event.type + " at " + event.currentTarget);

alert("Coordinates: " + event.clientX + ":" + event.clientY);

};

</script>

Some properties of event object:

**event.type**

Event type, here it’s "click".

**event.currentTarget**

Element that handled the event. That’s exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.

**event.clientX / event.clientY**

Window-relative coordinates of the cursor, for pointer events.

There are more properties. Many of them depend on the event type: keyboard events have one set of properties, pointer events – another one, we’ll study them later when as we move on to the details of different events.

**The event object is also available in HTML handlers**

If we assign a handler in HTML, we can also use the event object, like this:

<input type="button" onclick="alert(event.type)" value="Event type">

That’s possible because when the browser reads the attribute, it creates a handler like this: function(event) { alert(event.type) }. That is: its first argument is called "event", and the body is taken from the attribute.

## [Object handlers: handleEvent](https://javascript.info/introduction-browser-events" \l "object-handlers-handleevent)

We can assign not just a function, but an object as an event handler using addEventListener. When an event occurs, its handleEvent method is called.

For instance:

<button id="elem">Click me</button>

<script>

let obj = {

handleEvent(event) {

alert(event.type + " at " + event.currentTarget);

}

};

elem.addEventListener('click', obj);

</script>

As we can see, when addEventListener receives an object as the handler, it calls obj.handleEvent(event) in case of an event.

We could also use objects of a custom class, like this:

<button id="elem">Click me</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Mouse button pressed";

break;

case 'mouseup':

elem.innerHTML += "...and released.";

break;

}

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Here the same object handles both events. Please note that we need to explicitly setup the events to listen using addEventListener. The menu object only gets mousedown and mouseup here, not any other types of events.

The method handleEvent does not have to do all the job by itself. It can call other event-specific methods instead, like this:

<button id="elem">Click me</button>

<script>

class Menu {

handleEvent(event) {

// mousedown -> onMousedown

let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);

this[method](event);

}

onMousedown() {

elem.innerHTML = "Mouse button pressed";

}

onMouseup() {

elem.innerHTML += "...and released.";

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Now event handlers are clearly separated, that may be easier to support.

## [Summary](https://javascript.info/introduction-browser-events#summary)

There are 3 ways to assign event handlers:

1. HTML attribute: onclick="...".
2. DOM property: elem.onclick = function.
3. Methods: elem.addEventListener(event, handler[, phase]) to add, removeEventListener to remove.

HTML attributes are used sparingly, because JavaScript in the middle of an HTML tag looks a little bit odd and alien. Also can’t write lots of code in there.

DOM properties are ok to use, but we can’t assign more than one handler of the particular event. In many cases that limitation is not pressing.

The last way is the most flexible, but it is also the longest to write. There are few events that only work with it, for instance transitionend and DOMContentLoaded (to be covered). Also addEventListener supports objects as event handlers. In that case the method handleEvent is called in case of the event.

No matter how you assign the handler – it gets an event object as the first argument. That object contains the details about what’s happened.

## Tasks

<https://javascript.info/introduction-browser-events#tasks>

# JavaScript – Event Propagation – Bubbling and Capturing

Let’s start with an example.

This handler is assigned to <div>, but also runs if you click any nested tag like <em> or <code>:

<div onclick="alert('The handler!')">

<em>If you click on <code>EM</code>, the handler on <code>DIV</code> runs.</em>

</div>

Isn’t it a bit strange? Why does the handler on <div> run if the actual click was on <em>?

## [Bubbling](https://javascript.info/bubbling-and-capturing" \l "bubbling)

The bubbling principle is simple.

**When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.**

Let’s say we have 3 nested elements FORM > DIV > P with a handler on each of them:

<style>

body \* {

margin: 10px;

border: 1px solid blue;

}

</style>

<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

</div>

</form>

A click on the inner <p> first runs onclick:

1. On that <p>.
2. Then on the outer <div>.
3. Then on the outer <form>.
4. And so on upwards till the document object.

So, if we click on <p>, then we’ll see 3 alerts: p → div → form.

The process is called “bubbling”, because events “bubble” from the inner element up through parents like a bubble in the water.

**Almost all events bubble.**

The key word in this phrase is “almost”.

For instance, a focus event does not bubble. There are other examples too, we’ll meet them. But still it’s an exception, rather than a rule, most events do bubble.

## [event.target](https://javascript.info/bubbling-and-capturing" \l "event-target)

A handler on a parent element can always get the details about where it actually happened.

**The most deeply nested element that caused the event is called a target element, accessible as event.target.**

Note the differences from this (=event.currentTarget):

* event.target – is the “target” element that initiated the event, it doesn’t change through the bubbling process.
* this – is the “current” element, the one that has a currently running handler on it.

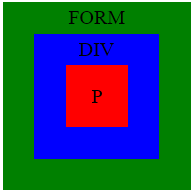
For instance, if we have a single handler form.onclick, then it can “catch” all clicks inside the form. No matter where the click happened, it bubbles up to <form> and runs the handler.

In form.onclick handler:

* this (=event.currentTarget) is the <form> element, because the handler runs on it.
* event.target is the actual element inside the form that was clicked.

**Check it out:**

**Result**: A click shows both event.target and this to compare:



**script.js:**

form.onclick = function(event) {

event.target.style.backgroundColor = 'yellow';

// chrome needs some time to paint yellow

setTimeout(() => {

alert("target = " + event.target.tagName + ", this=" + this.tagName);

event.target.style.backgroundColor = ''

}, 100);

};

**example.css:**

form {

background-color: green;

position: relative;

width: 150px;

height: 150px;

text-align: center;

cursor: pointer;

}

div {

background-color: blue;

position: absolute;

top: 25px;

left: 25px;

width: 100px;

height: 100px;

}

p {

background-color: red;

position: absolute;

top: 25px;

left: 25px;

width: 50px;

height: 50px;

line-height: 50px;

margin: 0;

}

body {

line-height: 25px;

font-size: 16px;

}

**index.html:**

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<link rel="stylesheet" href="example.css">

</head>

<body>

A click shows both <code>event.target</code> and <code>this</code> to compare:

<form id="form">FORM

<div>DIV

<p>P</p>

</div>

</form>

<script src="script.js"></script>

</body>

</html>

It’s possible that event.target could equal this – it happens when the click is made directly on the <form> element.

## [Stopping bubbling](https://javascript.info/bubbling-and-capturing" \l "stopping-bubbling)

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path.

But any handler may decide that the event has been fully processed and stop the bubbling.

The method for it is event.stopPropagation().

For instance, here body.onclick doesn’t work if you click on <button>:

<body onclick="alert(`the bubbling doesn't reach here`)">

<button onclick="event.stopPropagation()">Click me</button>

</body>

**event.stopImmediatePropagation()**

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.

In other words, event.stopPropagation() stops the move upwards, but on the current element all other handlers will run.

To stop the bubbling and prevent handlers on the current element from running, there’s a method event.stopImmediatePropagation(). After it no other handlers execute.

**Don’t stop bubbling without a need!**

Bubbling is convenient. Don’t stop it without a real need: obvious and architecturally well thought out.

Sometimes event.stopPropagation() creates hidden pitfalls that later may become problems.

For instance:

1. We create a nested menu. Each submenu handles clicks on its elements and calls stopPropagation so that the outer menu won’t trigger.
2. Later we decide to catch clicks on the whole window, to track users’ behavior (where people click). Some analytic systems do that. Usually the code uses document.addEventListener('click'…) to catch all clicks.
3. Our analytic won’t work over the area where clicks are stopped by stopPropagation. Sadly, we’ve got a “dead zone”.

There’s usually no real need to prevent the bubbling. A task that seemingly requires that may be solved by other means. One of them is to use custom events, we’ll cover them later. Also we can write our data into the event object in one handler and read it in another one, so we can pass to handlers on parents information about the processing below.

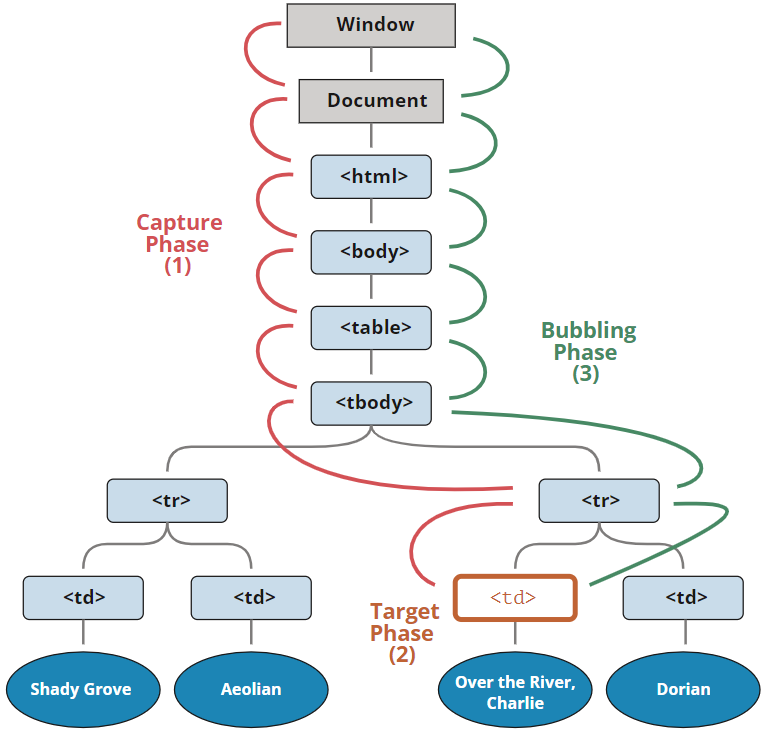
## [Capturing](https://javascript.info/bubbling-and-capturing" \l "capturing)

There’s another phase of event processing called “capturing”. It is rarely used in real code, but sometimes can be useful.

The standard [DOM Events](http://www.w3.org/TR/DOM-Level-3-Events/) describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

Here’s the picture, taken from the specification, of the capturing (1), target (2) and bubbling (3) phases for a click event on a <td> inside a table:



That is: for a click on <td> the event first goes through the ancestors chain down to the element (capturing phase), then it reaches the target and triggers there (target phase), and then it goes up (bubbling phase), calling handlers on its way.

Until now, we only talked about bubbling, because the capturing phase is rarely used.

In fact, the capturing phase was invisible for us, because handlers added using on<event>-property or using HTML attributes or using two-argument addEventListener(event, handler) don’t know anything about capturing, they only run on the 2nd and 3rd phases.

To catch an event on the capturing phase, we need to set the handler capture option to true:

elem.addEventListener(..., {capture: true})

// or, just "true" is an alias to {capture: true}

elem.addEventListener(..., true)

There are two possible values of the capture option:

* If it’s false (default), then the handler is set on the bubbling phase.
* If it’s true, then the handler is set on the capturing phase.

Note that while formally there are 3 phases, the 2nd phase (“target phase”: the event reached the element) is not handled separately: handlers on both capturing and bubbling phases trigger at that phase.

Let’s see both capturing and bubbling in action:

<style>

body \* {

margin: 10px;

border: 1px solid blue;

}

</style>

<form>FORM

<div>DIV

<p>P</p>

</div>

</form>

<script>

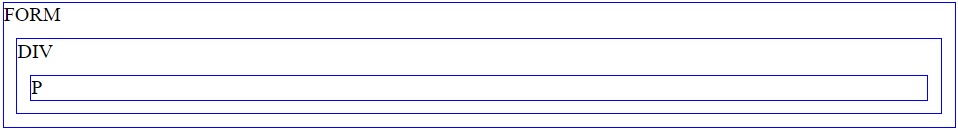
for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Capturing: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Bubbling: ${elem.tagName}`));

}

</script>



The code sets click handlers on every element in the document to see which ones are working.

If you click on <p>, then the sequence is:

1. HTML → BODY → FORM → DIV -> P (capturing phase, the first listener):
2. P → DIV → FORM → BODY → HTML (bubbling phase, the second listener).

Please note, the P shows up twice, because we’ve set two listeners: capturing and bubbling. The target triggers at the end of the first and at the beginning of the second phase.

There’s a property event.eventPhase that tells us the number of the phase on which the event was caught. But it’s rarely used, because we usually know it in the handler.

**To remove the handler, removeEventListener needs the same phase**

If we addEventListener(..., true), then we should mention the same phase in removeEventListener(..., true) to correctly remove the handler.

**Listeners on the same element and same phase run in their set order**

If we have multiple event handlers on the same phase, assigned to the same element with addEventListener, they run in the same order as they are created:

elem.addEventListener("click", e => alert(1)); // guaranteed to trigger first

elem.addEventListener("click", e => alert(2));

**The event.stopPropagation() during the capturing also prevents the bubbling**

The event.stopPropagation() method and its sibling event.stopImmediatePropagation() can also be called on the capturing phase. Then not only the futher capturing is stopped, but the bubbling as well.

In other words, normally the event goes first down (“capturing”) and then up (“bubbling”). But if event.stopPropagation() is called during the capturing phase, then the event travel stops, no bubbling will occur.

## [Summary](https://javascript.info/bubbling-and-capturing#summary)

When an event happens – the most nested element where it happens gets labeled as the “target element” (event.target).

* Then the event moves down from the document root to event.target, calling handlers assigned with addEventListener(..., true) on the way (true is a shorthand for {capture: true}).
* Then handlers are called on the target element itself.
* Then the event bubbles up from event.target to the root, calling handlers assigned using on<event>, HTML attributes and addEventListener without the 3rd argument or with the 3rd argument false/{capture:false}.

Each handler can access event object properties:

* event.target – the deepest element that originated the event.
* event.currentTarget (=this) – the current element that handles the event (the one that has the handler on it)
* event.eventPhase – the current phase (capturing=1, target=2, bubbling=3).

Any event handler can stop the event by calling event.stopPropagation(), but that’s not recommended, because we can’t really be sure we won’t need it above, maybe for completely different things.

The capturing phase is used very rarely, usually we handle events on bubbling. And there’s a logical explanation for that.

In real world, when an accident happens, local authorities react first. They know best the area where it happened. Then higher-level authorities if needed.

The same for event handlers. The code that set the handler on a particular element knows maximum details about the element and what it does. A handler on a particular <td> may be suited for that exactly <td>, it knows everything about it, so it should get the chance first. Then its immediate parent also knows about the context, but a little bit less, and so on till the very top element that handles general concepts and runs the last one.

# JavaScript – Storage – Cookies, document.cookie

<https://javascript.info/cookie>

Cookies are small strings of data that are stored directly in the browser. They are a part of the HTTP protocol, defined by the [RFC 6265](https://tools.ietf.org/html/rfc6265) specification.

Cookies are usually set by a web-server using the response Set-Cookie HTTP-header. Then, the browser automatically adds them to (almost) every request to the same domain using the Cookie HTTP-header.

One of the most widespread use cases is authentication:

1. Upon sign in, the server uses the Set-Cookie HTTP-header in the response to set a cookie with a unique “session identifier”.
2. Next time when the request is sent to the same domain, the browser sends the cookie over the net using the Cookie HTTP-header.
3. So the server knows who made the request.

We can also access cookies from the browser, using document.cookie property.

# JavaScript – Storage – LocalStorage, sessionStorage

Web storage objects localStorage and sessionStorage allow to save key/value pairs in the browser.

What’s interesting about them is that the data survives a page refresh (for sessionStorage) and even a full browser restart (for localStorage). We’ll see that very soon.

We already have cookies. Why additional objects?

* Unlike cookies, web storage objects are not sent to server with each request. Because of that, we can store much more. Most modern browsers allow at least 5 megabytes of data (or more) and have settings to configure that.
* Also unlike cookies, the server can’t manipulate storage objects via HTTP headers. Everything’s done in JavaScript.
* The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or subdomains infer different storage objects, they can’t access data from each other.

Both storage objects provide same methods and properties:

* setItem(key, value) – store key/value pair.
* getItem(key) – get the value by key.
* removeItem(key) – remove the key with its value.
* clear() – delete everything.
* key(index) – get the key on a given position.
* length – the number of stored items.

As you can see, it’s like a Map collection (setItem/getItem/removeItem), but also allows access by index with key(index).

Let’s see how it works.

## [localStorage demo](https://javascript.info/localstorage" \l "localstorage-demo)

The main features of localStorage are:

* Shared between all tabs and windows from the same origin.
* The data does not expire. It remains after the browser restart and even OS reboot.

For instance, if you run this code…

localStorage.setItem('test', 1);

…And close/open the browser or just open the same page in a different window, then you can get it like this:

alert( localStorage.getItem('test') ); // 1

We only have to be on the same origin (domain/port/protocol), the url path can be different.

The localStorage is shared between all windows with the same origin, so if we set the data in one window, the change becomes visible in another one.

## [Object-like access](https://javascript.info/localstorage" \l "object-like-access)

We can also use a plain object way of getting/setting keys, like this:

// set key

localStorage.test = 2;

// get key

alert( localStorage.test ); // 2

// remove key

delete localStorage.test;

alert( localStorage.test ); // undefined

That’s allowed for historical reasons, and mostly works, but generally not recommended, because:

1. If the key is user-generated, it can be anything, like length or toString, or another built-in method of localStorage. In that case getItem/setItem work fine, while object-like access fails:

let key = 'length';

localStorage[key] = 5; // Error, can't assign length

1. There’s a storage event, it triggers when we modify the data. That event does not happen for object-like access. We’ll see that later in this chapter.

## [Looping over keys](https://javascript.info/localstorage" \l "looping-over-keys)

As we’ve seen, the methods provide “get/set/remove by key” functionality. But how to get all saved values or keys?

Unfortunately, storage objects are not iterable.

One way is to loop over them as over an array:

for(let i=0; i<localStorage.length; i++) {

let key = localStorage.key(i);

alert(`${key}: ${localStorage.getItem(key)}`);

}

Another way is to use for key in localStorage loop, just as we do with regular objects.

It iterates over keys, but also outputs few built-in fields that we don’t need:

// bad try

for(let key in localStorage) {

alert(key); // shows getItem, setItem and other built-in stuff

}

…So we need either to filter fields from the prototype with hasOwnProperty check:

for(let key in localStorage) {

if (!localStorage.hasOwnProperty(key)) {

continue; // skip keys like "setItem", "getItem" etc

}

alert(`${key}: ${localStorage.getItem(key)}`);

}

…Or just get the “own” keys with Object.keys and then loop over them if needed:

let keys = Object.keys(localStorage);

for(let key of keys) {

alert(`${key}: ${localStorage.getItem(key)}`);

}

The latter works, because Object.keys only returns the keys that belong to the object, ignoring the prototype.

## [Strings only](https://javascript.info/localstorage" \l "strings-only)

Please note that both key and value must be strings.

If were any other type, like a number, or an object, it gets converted to string automatically:

localStorage.user = {name: "John"};

alert(localStorage.user); // [object Object]

We can use JSON to store objects though:

localStorage.user = JSON.stringify({name: "John"});

// sometime later

let user = JSON.parse( localStorage.user );

alert( user.name ); // John

Also it is possible to stringify the whole storage object, e.g. for debugging purposes:

// added formatting options to JSON.stringify to make the object look nicer

alert( JSON.stringify(localStorage, null, 2) );

## [sessionStorage](https://javascript.info/localstorage" \l "sessionstorage)

The sessionStorage object is used much less often than localStorage.

Properties and methods are the same, but it’s much more limited:

* The sessionStorage exists only within the current browser tab.
  + Another tab with the same page will have a different storage.
  + But it is shared between iframes in the same tab (assuming they come from the same origin).
* The data survives page refresh, but not closing/opening the tab.

Let’s see that in action.

Run this code…

sessionStorage.setItem('test', 1);

…Then refresh the page. Now you can still get the data:

alert( sessionStorage.getItem('test') ); // after refresh: 1

…But if you open the same page in another tab, and try again there, the code above returns null, meaning “nothing found”.

That’s exactly because sessionStorage is bound not only to the origin, but also to the browser tab. For that reason, sessionStorage is used sparingly.

## [Storage event](https://javascript.info/localstorage" \l "storage-event)

When the data gets updated in localStorage or sessionStorage, [storage](https://html.spec.whatwg.org/multipage/webstorage.html#the-storageevent-interface) event triggers, with properties:

* key – the key that was changed (null if .clear() is called).
* oldValue – the old value (null if the key is newly added).
* newValue – the new value (null if the key is removed).
* url – the url of the document where the update happened.
* storageArea – either localStorage or sessionStorage object where the update happened.

The important thing is: the event triggers on all window objects where the storage is accessible, except the one that caused it.

Let’s elaborate.

Imagine, you have two windows with the same site in each. So localStorage is shared between them.

You might want to open this page in two browser windows to test the code below.

If both windows are listening for window.onstorage, then each one will react on updates that happened in the other one.

// triggers on updates made to the same storage from other documents

window.onstorage = event => { // can also use window.addEventListener('storage', event => {

if (event.key != 'now') return;

alert(event.key + ':' + event.newValue + " at " + event.url);

};

localStorage.setItem('now', Date.now());

Please note that the event also contains: event.url – the url of the document where the data was updated.

Also, event.storageArea contains the storage object – the event is the same for both sessionStorage and localStorage, so event.storageArea references the one that was modified. We may even want to set something back in it, to “respond” to a change.

**That allows different windows from the same origin to exchange messages.**

Modern browsers also support [Broadcast channel API](https://developer.mozilla.org/en-US/docs/Web/api/Broadcast_Channel_API), the special API for same-origin inter-window communication, it’s more full featured, but less supported. There are libraries that polyfill that API, based on localStorage, that make it available everywhere.

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

Web storage objects localStorage and sessionStorage allow to store key/value in the browser.

* Both key and value must be strings.
* The limit is 5mb+, depends on the browser.
* They do not expire.
* The data is bound to the origin (domain/port/protocol).

| **localStorage** | **sessionStorage** |
| --- | --- |
| Shared between all tabs and windows with the same origin | Visible within a browser tab, including iframes from the same origin |
| Survives browser restart | Survives page refresh (but not tab close) |

**API:**

* setItem(key, value) – store key/value pair.
* getItem(key) – get the value by key.
* removeItem(key) – remove the key with its value.
* clear() – delete everything.
* key(index) – get the key number index.
* length – the number of stored items.
* Use Object.keys to get all keys.
* We access keys as object properties, in that case storage event isn’t triggered.

**Storage event:**

* Triggers on setItem, removeItem, clear calls.
* Contains all the data about the operation (key/oldValue/newValue), the document url and the storage object storageArea.
* Triggers on all window objects that have access to the storage except the one that generated it (within a tab for sessionStorage, globally for localStorage).

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

### [Autosave a form field](https://javascript.info/localstorage" \l "autosave-a-form-field)

Create a textarea field that “autosaves” its value on every change.

So, if the user accidentally closes the page, and opens it again, he’ll find his unfinished input at place.

Like this:



**Solution**

<!doctype html>

<textarea style="width:200px; height: 60px;" id="area" placeholder="Write here"></textarea>

<br>

<button onclick="localStorage.removeItem('area');area.value=''">Clear</button>

<script>

    area.value = localStorage.getItem('area');

    area.oninput = () => {

      localStorage.setItem('area', area.value)

    };

</script>

# JavaScript – Promises and Callbacks

<https://javascript.info/callbacks>

<https://javascript.info/promise-basics>

## Introduction to Callbacks

Many functions are provided by JavaScript host environments that allow you to schedule asynchronous actions. In other words, actions that we initiate now, but they finish later.

For instance, one such function is the setTimeout function.

There are other real-world examples of asynchronous actions, e.g. loading scripts and modules (we’ll cover them in later chapters).

Take a look at the function loadScript(src), that loads a script with the given src:

function loadScript(src) {

// creates a <script> tag and append it to the page

// this causes the script with given src to start loading and run when complete

let script = document.createElement('script');

script.src = src;

document.head.append(script);

}

It inserts into the document a new, dynamically created, tag <script src="…"> with the given src. The browser automatically starts loading it and executes when complete.

We can use this function like this:

// load and execute the script at the given path

loadScript('/my/script.js');

The script is executed “asynchronously”, as it starts loading now, but runs later, when the function has already finished.

If there’s any code below loadScript(…), it doesn’t wait until the script loading finishes.

loadScript('/my/script.js');

// the code below loadScript

// doesn't wait for the script loading to finish

// ...

Let’s say we need to use the new script as soon as it loads. It declares new functions, and we want to run them.

But if we do that immediately after the loadScript(…) call, that wouldn’t work:

loadScript('/my/script.js'); // the script has "function newFunction() {…}"

newFunction(); // no such function!

Naturally, the browser probably didn’t have time to load the script. As of now, the loadScript function doesn’t provide a way to track the load completion. The script loads and eventually runs, that’s all. But we’d like to know when it happens, to use new functions and variables from that script.

Let’s add a callback function as a second argument to loadScript that should execute when the script loads:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

The onload event is described in the article [Resource loading: onload and onerror](https://javascript.info/onload-onerror#loading-a-script), it basically executes a function after the script is loaded and executed.

Now if we want to call new functions from the script, we should write that in the callback:

loadScript('/my/script.js', function() {

// the callback runs after the script is loaded

newFunction(); // so now it works

...

});

That’s the idea: the second argument is a function (usually anonymous) that runs when the action is completed.

Here’s a runnable example with a real script:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Cool, the script ${script.src} is loaded`);

alert( \_ ); // \_ is a function declared in the loaded script

});

That’s called a “callback-based” style of asynchronous programming. A function that does something asynchronously should provide a callback argument where we put the function to run after it’s complete.

Here we did it in loadScript, but of course it’s a general approach.

### [Callback in callback](https://javascript.info/callbacks" \l "callback-in-callback)

How can we load two scripts sequentially: the first one, and then the second one after it?

The natural solution would be to put the second loadScript call inside the callback, like this:

loadScript('/my/script.js', function(script) {

alert(`Cool, the ${script.src} is loaded, let's load one more`);

loadScript('/my/script2.js', function(script) {

alert(`Cool, the second script is loaded`);

});

});

After the outer loadScript is complete, the callback initiates the inner one.

What if we want one more script…?

loadScript('/my/script.js', function(script) {

loadScript('/my/script2.js', function(script) {

loadScript('/my/script3.js', function(script) {

// ...continue after all scripts are loaded

});

});

});

So, every new action is inside a callback. That’s fine for few actions, but not good for many, so we’ll see other variants soon.

### [Handling errors](https://javascript.info/callbacks" \l "handling-errors)

In the above examples we didn’t consider errors. What if the script loading fails? Our callback should be able to react on that.

Here’s an improved version of loadScript that tracks loading errors:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Script load error for ${src}`));

document.head.append(script);

}

It calls callback(null, script) for successful load and callback(error) otherwise.

The usage:

loadScript('/my/script.js', function(error, script) {

if (error) {

// handle error

} else {

// script loaded successfully

}

});

Once again, the recipe that we used for loadScript is actually quite common. It’s called the “error-first callback” style.

The convention is:

1. The first argument of the callback is reserved for an error if it occurs. Then callback(err) is called.
2. The second argument (and the next ones if needed) are for the successful result. Then callback(null, result1, result2…) is called.

So, the single callback function is used both for reporting errors and passing back results.

### [Pyramid of Doom](https://javascript.info/callbacks" \l "pyramid-of-doom)

At first glance, it looks like a viable approach to asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.

But for multiple asynchronous actions that follow one after another, we’ll have code like this:

loadScript('1.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('2.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('3.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...continue after all scripts are loaded (\*)

}

});

}

});

}

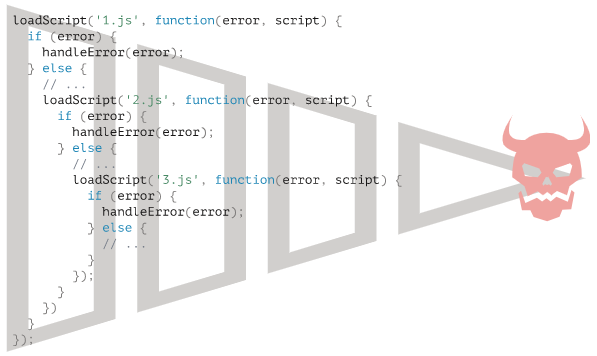
});

In the code above:

1. We load 1.js, then if there’s no error…
2. We load 2.js, then if there’s no error…
3. We load 3.js, then if there’s no error – do something else (\*).

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of ... that may include more loops, conditional statements and so on.

That’s sometimes called “callback hell” or “pyramid of doom.”



The “pyramid” of nested calls grows to the right with every asynchronous action. Soon it spirals out of control.

So, this way of coding isn’t very good.

We can try to alleviate the problem by making every action a standalone function, like this:

loadScript('1.js', step1);

function step1(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('2.js', step2);

}

}

function step2(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('3.js', step3);

}

}

function step3(error, script) {

if (error) {

handleError(error);

} else {

// ...continue after all scripts are loaded (\*)

}

}

See? It does the same thing, and there’s no deep nesting now because we made every action a separate top-level function.

It works, but the code looks like a torn apart spreadsheet. It’s difficult to read, and you probably noticed that one needs to eye-jump between pieces while reading it. That’s inconvenient, especially if the reader is not familiar with the code and doesn’t know where to eye-jump.

Also, the functions named step\* are all of single use, they are created only to avoid the “pyramid of doom.” No one is going to reuse them outside of the action chain. So, there’s a bit of namespace cluttering here.

## Promise

Imagine that you’re a top singer, and fans ask day and night for your upcoming song.

To get some relief, you promise to send it to them when it’s published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can’t publish the song, they will still be notified.

Everyone is happy: you, because the people don’t crowd you anymore, and fans, because they won’t miss the song.

This is a real-life analogy for things we often have in programming:

1. A “producing code” that does something and takes time. For instance, some code that loads the data over a network. That’s a “singer”.
2. A “consuming code” that wants the result of the “producing code” once it’s ready. Many functions may need that result. These are the “fans”.
3. A promise is a special JavaScript object that links the “producing code” and the “consuming code” together. In terms of our analogy: this is the “subscription list”. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

The analogy isn’t terribly accurate, because JavaScript promises are more complex than a simple subscription list: they have additional features and limitations. But it’s fine to begin with.

The constructor syntax for a promise object is:

let promise = new Promise(function(resolve, reject) {

// executor (the producing code, "singer")

});

The function passed to new Promise is called the executor. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the “singer”.

Its arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor.

When the executor obtains the result, be it soon or late, doesn’t matter, it should call one of these callbacks:

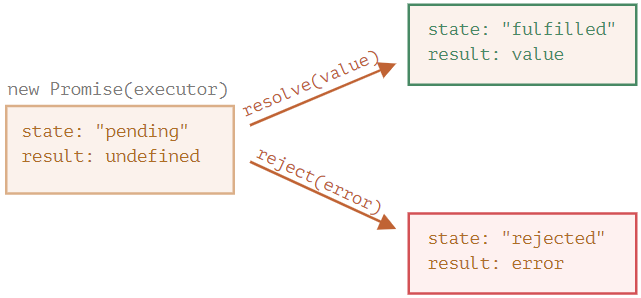
* resolve(value) — if the job is finished successfully, with result value.
* reject(error) — if an error has occurred, error is the error object.

So, to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt, it calls resolve if it was successful or reject if there was an error.

The promise object returned by the new Promise constructor has these internal properties:

* state — initially "pending", then changes to either "fulfilled" when resolve is called or "rejected" when reject is called.
* result — initially undefined, then changes to value when resolve(value) is called or error when reject(error) is called.

So, the executor eventually moves promise to one of these states:



Later we’ll see how “fans” can subscribe to these changes.

Here’s an example of a promise constructor and a simple executor function with “producing code” that takes time (via setTimeout):

let promise = new Promise(function(resolve, reject) {

// the function is executed automatically when the promise is constructed

// after 1 second signal that the job is done with the result "done"

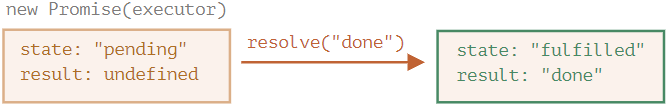
setTimeout(() => resolve("done"), 1000);

});

We can see two things by running the code above:

1. The executor is called automatically and immediately (by new Promise).
2. The executor receives two arguments: resolve and reject. These functions are pre-defined by the JavaScript engine, so we don’t need to create them. We should only call one of them when ready.

After one second of “processing”, the executor calls resolve("done") to produce the result. This changes the state of the promise object:



That was an example of a successful job completion, a “fulfilled promise”.

And now an example of the executor rejecting the promise with an error:

let promise = new Promise(function(resolve, reject) {

// after 1 second signal that the job is finished with an error

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

The call to reject(...) moves the promise object to "rejected" state:



To summarize, the executor should perform a job (usually something that takes time) and then call resolve or reject to change the state of the corresponding promise object.

A promise that is either resolved or rejected is called “settled”, as opposed to an initially “pending” promise.

**There can be only a single result or an error**

The executor should call only one resolve or one reject. Any state change is final.

All further calls of resolve and reject are ignored:

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // ignored

setTimeout(() => resolve("…")); // ignored

});

The idea is that a job done by the executor may have only one result or an error.

Also, resolve/reject expect only one argument (or none) and will ignore additional arguments.

**Reject with Error objects**

In case something goes wrong, the executor should call reject. That can be done with any type of argument (just like resolve). But it is recommended to use Error objects (or objects that inherit from Error). The reasoning for that will soon become apparent.

**Immediately calling resolve/reject**

In practice, an executor usually does something asynchronously and calls resolve/reject after some time, but it doesn’t have to. We also can call resolve or reject immediately, like this:

let promise = new Promise(function(resolve, reject) {

// not taking our time to do the job

resolve(123); // immediately give the result: 123

});

For instance, this might happen when we start to do a job but then see that everything has already been completed and cached.

That’s fine. We immediately have a resolved promise.

**The state and result are internal**

The properties state and result of the Promise object are internal. We can’t directly access them. We can use the methods .then/.catch/.finally for that. They are described below.

### [Consumers: then, catch](https://javascript.info/promise-basics" \l "consumers-then-catch)

A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using the methods .then and .catch.

#### [then](https://javascript.info/promise-basics" \l "then)

The most important, fundamental one is .then.

The syntax is:

promise.then(

function(result) { /\* handle a successful result \*/ },

function(error) { /\* handle an error \*/ }

);

The first argument of .then is a function that runs when the promise is resolved and receives the result.

The second argument of .then is a function that runs when the promise is rejected and receives the error.

For instance, here’s a reaction to a successfully resolved promise:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve runs the first function in .then

promise.then(

result => alert(result), // shows "done!" after 1 second

error => alert(error) // doesn't run

);

The first function was executed.

And in the case of a rejection, the second one:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// reject runs the second function in .then

promise.then(

result => alert(result), // doesn't run

error => alert(error) // shows "Error: Whoops!" after 1 second

);

If we’re interested only in successful completions, then we can provide only one function argument to .then:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // shows "done!" after 1 second

#### [catch](https://javascript.info/promise-basics" \l "catch)

If we’re interested only in errors, then we can use null as the first argument: .then(null, errorHandlingFunction). Or we can use .catch(errorHandlingFunction), which is exactly the same:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// .catch(f) is the same as promise.then(null, f)

promise.catch(alert); // shows "Error: Whoops!" after 1 second

The call .catch(f) is a complete analog of .then(null, f), it’s just a shorthand.

### [Cleanup: finally](https://javascript.info/promise-basics" \l "cleanup-finally)

Just like there’s a finally clause in a regular try {...} catch {...}, there’s finally in promises.

The call .finally(f) is similar to .then(f, f) in the sense that f runs always, when the promise is settled: be it resolve or reject.

The idea of finally is to set up a handler for performing cleanup/finalizing after the previous operations are complete.

E.g. stopping loading indicators, closing no longer needed connections, etc.

Think of it as a party finisher. No matter was a party good or bad, how many friends were in it, we still need (or at least should) do a cleanup after it.

The code may look like this:

new Promise((resolve, reject) => {

/\* do something that takes time, and then call resolve or maybe reject \*/

})

// runs when the promise is settled, doesn't matter successfully or not

.finally(() => stop loading indicator)

// so the loading indicator is always stopped before we go on

.then(result => show result, err => show error)

Please note that finally(f) isn’t exactly an alias of then(f,f) though.

There are important differences:

1. A finally handler has no arguments. In finally we don’t know whether the promise is successful or not. That’s all right, as our task is usually to perform “general” finalizing procedures.

Please take a look at the example above: as you can see, the finally handler has no arguments, and the promise outcome is handled by the next handler.

1. A finally handler “passes through” the result or error to the next suitable handler.

For instance, here the result is passed through finally to then:

new Promise((resolve, reject) => {

setTimeout(() => resolve("value"), 2000);

})

.finally(() => alert("Promise ready")) // triggers first

.then(result => alert(result)); // <-- .then shows "value"

As you can see, the value returned by the first promise is passed through finally to the next then.

That’s very convenient, because finally is not meant to process a promise result. As said, it’s a place to do generic cleanup, no matter what the outcome was.

And here’s an example of an error, for us to see how it’s passed through finally to catch:

new Promise((resolve, reject) => {

throw new Error("error");

})

.finally(() => alert("Promise ready")) // triggers first

.catch(err => alert(err)); // <-- .catch shows the error

1. A finally handler also shouldn’t return anything. If it does, the returned value is silently ignored.

The only exception to this rule is when a finally handler throws an error. Then this error goes to the next handler, instead of any previous outcome.

To summarize:

* A finally handler doesn’t get the outcome of the previous handler (it has no arguments). This outcome is passed through instead, to the next suitable handler.
* If a finally handler returns something, it’s ignored.
* When finally throws an error, then the execution goes to the nearest error handler.

These features are helpful and make things work just the right way if we use finally how it’s supposed to be used: for generic cleanup procedures.

**We can attach handlers to settled promises**

If a promise is pending, .then/catch/finally handlers wait for its outcome.

Sometimes, it might be that a promise is already settled when we add a handler to it.

In such case, these handlers just run immediately:

// the promise becomes resolved immediately upon creation

let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (shows up right now)

Note that this makes promises more powerful than the real life “subscription list” scenario. If the singer has already released their song and then a person signs up on the subscription list, they probably won’t receive that song. Subscriptions in real life must be done prior to the event.

Promises are more flexible. We can add handlers any time: if the result is already there, they just execute.

### [Example: loadScript](https://javascript.info/promise-basics" \l "loadscript)

Next, let’s see more practical examples of how promises can help us write asynchronous code.

We’ve got the loadScript function for loading a script from the previous chapter.

Here’s the callback-based variant, just to remind us of it:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Script load error for ${src}`));

document.head.append(script);

}

Let’s rewrite it using Promises.

The new function loadScript will not require a callback. Instead, it will create and return a Promise object that resolves when the loading is complete. The outer code can add handlers (subscribing functions) to it using .then:

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Script load error for ${src}`));

document.head.append(script);

});

}

Usage:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} is loaded!`),

error => alert(`Error: ${error.message}`)

);

promise.then(script => alert('Another handler...'));

We can immediately see a few benefits over the callback-based pattern:

| **Promises** | **Callbacks** |
| --- | --- |
| Promises allow us to do things in the natural order. First, we run loadScript(script), and .then we write what to do with the result. | We must have a callback function at our disposal when calling loadScript(script, callback). In other words, we must know what to do with the result before loadScript is called. |
| We can call .then on a Promise as many times as we want. Each time, we’re adding a new “fan”, a new subscribing function, to the “subscription list”. More about this in the next chapter: [Promises chaining](https://javascript.info/promise-chaining). | There can be only one callback. |

So, promises give us better code flow and flexibility. But there’s more. We’ll see that in the next chapters.

### [Tasks](https://javascript.info/promise-basics#tasks)

#### [Re-resolve a promise?](https://javascript.info/promise-basics" \l "re-resolve-a-promise)

What’s the output of the code below?

let promise = new Promise(function(resolve, reject) {

resolve(1);

setTimeout(() => resolve(2), 1000);

});

promise.then(alert);

**Solution**

The output is: 1.

The second call to resolve is ignored, because only the first call of reject/resolve is taken into account. Further calls are ignored.

#### [Delay with a promise](https://javascript.info/promise-basics#delay-with-a-promise)

The built-in function setTimeout uses callbacks. Create a promise-based alternative.

The function delay(ms) should return a promise. That promise should resolve after ms milliseconds, so that we can add .then to it, like this:

function delay(ms) {

// your code

}

delay(3000).then(() => alert('runs after 3 seconds'));

**Solution**

function delay(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

delay(3000).then(() => alert('runs after 3 seconds'));

Please note that in this task resolve is called without arguments. We don’t return any value from delay, just ensure the delay.

#### [Animated circle with promise](https://javascript.info/promise-basics#animated-circle-with-promise)

Rewrite the showCircle function in the solution of the task [Animated circle with callback](https://javascript.info/task/animate-circle-callback) so that it returns a promise instead of accepting a callback.

The new usage:

showCircle(150, 150, 100).then(div => {

div.classList.add('message-ball');

div.append("Hello, world!");

});

Take the solution of the task [Animated circle with callback](https://javascript.info/task/animate-circle-callback) as the base.

**Solution**

<!DOCTYPE html>

<html>

<head>

  <meta charset="utf-8">

  <style>

    .message-ball {

      font-size: 20px;

      line-height: 200px;

      text-align: center;

    }

    .circle {

      transition-property: width, height;

      transition-duration: 2s;

      position: fixed;

      transform: translateX(-50%) translateY(-50%);

      background-color: red;

      border-radius: 50%;

    }

  </style>

</head>

<body>

  <button onclick="go()">Click me</button>

  <script>

  function go() {

    showCircle(150, 150, 100).then(div => {

      div.classList.add('message-ball');

      div.append("Hello, world!");

    });

  }

  function showCircle(cx, cy, radius) {

    let div = document.createElement('div');

    div.style.width = 0;

    div.style.height = 0;

    div.style.left = cx + 'px';

    div.style.top = cy + 'px';

    div.className = 'circle';

    document.body.append(div);

    return new Promise(resolve => {

      setTimeout(() => {

        div.style.width = radius \* 2 + 'px';

        div.style.height = radius \* 2 + 'px';

        div.addEventListener('transitionend', function handler() {

          div.removeEventListener('transitionend', handler);

          resolve(div);

        });

      }, 0);

    })

  }

  </script>

</body>

</html>

# How Promises Work in JavaScript – A Comprehensive Beginner's Guide

<https://www.freecodecamp.org/news/guide-to-javascript-promises/>

**Solution**: <https://github.com/AjaySingala/ng-ts-node/tree/main/JavaScript%20Demos/Promises>

JavaScript has the ability to carry out asynchronous (or async) instructions. These instructions run in the background until they have finished processing.

Asynchronous instructions do not stop the JavaScript engine from actively accepting and processing more instructions. This is why JavaScript is non-blocking in nature.

There are a few asynchronous features in JavaScript, and one of them is **Promises**. To work with promises, you must adopt a special syntax that makes writing async instructions a lot more organized. Working with promises is a very useful skill every JavaScript developer should learn.

This article is an in-depth guide to promises in JavaScript. You are going to learn why JavaScript has promises, what a promise is, and how to work with it. You are also going to learn how to use async/await—a feature derived from promises—and what a job queue is.

## Why Should You Care about Promises?

Promises were not always part of JavaScript. Callbacks worked together with asynchronous functions to produce desired results in the past. A callback is any function that is a parameter of an async function, which the async function invokes to complete its operation.

To call an async function, you had to pass a callback as an argument like this:

function callback(result) {

// Use the result from the Async operation

}

randomAsyncOperation((response) => callback(response));

But callbacks had a huge problem. Demonstrating the problem makes understanding it easier.

Assume you had an asynchronous function that fetched data somewhere on the internet. This function should accept two callbacks, successCallback and failureCallback.

The successCallback would run if the operation was successful and the program found the appropriate resource. But the failureCallback would run if the operation was unsuccessful and could not find the resource.

function SuccessCallback(result) {

console.log("Resource found", result);

}

function failureCallback(error) {

console.error("Ooops. Something went wrong", error);

}

To run the async function, you had to pass the two callback functions as arguments:

fetchResource(url, successCallback, failureCallback)

Here, url is a variable that represents the location of the resource.

This code will run smoothly for now. You've taken care of both possible scenarios the function could run into. You have a callback for a successful operation and a callback for a failed operation.

Now assume you want to perform many other fetch operations, but each operation must be successful for the next one to run. This is useful if the data you need must come in a certain order and cannot be scattered.

For example, you might run into this situation if the result of the next operation depends on the result of the previous one.

In this case, your success callbacks would have their own success callbacks, which is important because you need to use the results if they come in.

fetchResource(

url,

function (result) {

// Do something with the result

fetchResource(

newUrl,

function (result) {

// Do something with the new result

fetchResource(

anotherUrl,

function (result) {

// Do something with the new result

},

failureCallback

);

},

failureCallback

);

},

failureCallback

);

From the example, you may notice a complication developing. You would have to keep nesting your success callbacks while repeating the failureCallback every time you call the async function.

These nested callbacks led to the [‘Callback Pyramid of Doom’](https://medium.com/dsc-srm/javascript-callback-hell-or-pyramid-of-doom-4f786d14b997) or callback hell, which can quickly become a nightmare. Could there be a more efficient way of handling situations like this?

JavaScript introduced Promises as part of [ES6 (ES2015)](https://262.ecma-international.org/6.0/#sec-promise-constructor) to solve this problem. It simplified working with callbacks and made for better syntax as you'll see shortly. Promises are now the foundation for most modern asynchronous operations developers use in JavaScript today.

## What is a Promise?



A promise is an assurance or guarantee that something will happen in the future. A person can promise another person a specific outcome or result. Promises are not limited to individuals, governments and organizations can also make promises. You have probably made a promise before.

With this assurance (promise) comes two possible outcomes–either fulfillment or failure. A promise is tied to an outcome that will show it is fulfilled. If that outcome does not happen, then the promise failed. A promise at the end must have one of these results.

In JavaScript, a Promise is an [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects) that will produce a single value some time in the future. If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed. The possible outcomes here are similar to that of promises in real life.

JavaScript promises can be in one of three possible states. These states indicate the progress of the promise. They are:

* pending: This is the default state of a defined promise
* fulfilled:  This is the state of a successful promise
* rejected: This is the state of a failed promise

A promise goes from pending to fulfilled, or from pending to rejected—‘fulfilled’ and ‘rejected’ indicate the end of a promise.

From now on, this article will refer to a 'promise' as the JavaScript object.

## How to Create a Promise in JavaScript

To create a promise, you need to create an instance object using the Promise constructor function. The Promise constructor function takes in one parameter. That parameter is a function that defines when to resolve the new promise, and optionally when to reject it.

const promise = new Promise((resolve, reject) => {

// Condition to resolve or reject the promise

});

For example, assume you want a promise to resolve after a timeout of two seconds. You can achieve this by writing it into the parameter of the constructor function.

const promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Done!"), 2000);

});

In promises, resolve is a function with an optional parameter representing the resolved value. Also, reject is a function with an optional parameter representing the reason why the promise failed. In the example above, the resolved value of the promise is the string 'Done!'.

Here is yet another example showing how you can resolve or reject a promise based on the conditions you set. In this example, the outcome of the promise is based on a random number the program generates.

const promise = new Promise((resolve, reject) => {

const num = Math.random();

if (num >= 0.5) {

resolve("Promise is fulfilled!");

} else {

reject("Promise failed!");

}

});

From these examples, you can see that you have control over when to resolve or reject your promise and can tie it to a certain condition. With that, you have learned how to create a promise in JavaScript.

## How to Attach a Callback to a Promise

To create a callback for a promise, you need to use the .then() method. This method takes in two callback functions. The first function runs if the promise is resolved, while the second function runs if the promise is rejected.

const promise = new Promise((resolve, reject) => {

const num = Math.random();

if (num >= 0.5) {

resolve("Promise is fulfilled!");

} else {

reject("Promise failed!");

}

});

function handleResolve(value) {

console.log(value);

}

function handleReject(reason) {

console.error(reason);

}

promise.then(handleResolve, handleReject);

// Promise is fulfilled!

// or

// Promise failed!

That is the way to handle the possible outcomes of your promise. Any unhandled errors in your promise will keep them in a rejected state at the end but handled errors makes the operation return a fulfilled promise.

It is possible to create an immediately resolved promise, and then attach a callback with the .then() method. You can also create an immediately rejected promise in the same way too.

Promise.resolve("Successful").then((result) => console.log(result));

// Successful

Promise.reject("Not successful").then((result) => console.log(result));

// Error: Uncaught (in promise)

The error in the rejected promise is because you need to define a separate callback to handle a rejected promise.

Promise.reject("Not successful").then(

() => {

/\*Empty Callback if Promise is fulfilled\*/

},

(reason) => console.error(reason)

);

// Not Successful

Now you have properly handled a rejected outcome.

Promises make it incredibly easy to chain asynchronous instructions. When you handle a promise with the .then() method, the operation always returns another promise. By employing this approach, you can eliminate the previously mentioned 'Callback Pyramid of Doom'.

Consider the code that previously caused the pyramid structure:

fetchResource(

url,

function (result) {

// Do something with the result

fetchResource(

newUrl,

function (result) {

// Do something with the new result

fetchResource(

anotherUrl,

function (result) {

// Do something with the new result

},

failureCallback

);

},

failureCallback

);

},

failureCallback

);

However, because .then() returns another promise, this is how to write the same instructions above with promises:

fetchResource(url)

.then(handleResult, failureCallback)

.then(handleNewResult, failureCallback)

.then(handleAnotherResult, failureCallback);

As you can see, calling promises does not require a nested syntax. You can even eliminate the repeated failureCallback to make the code a lot neater, which is something the upcoming section of the article will explore.

## How to Handle Errors in a Promise

To handle errors in Promises, use the .catch() method. If anything goes wrong with any of your promises, this method can catch the reason for that error.

Promise.reject(new Error()).catch((reason) => console.error(reason));

// Error

This time in our example, the error output is no longer ‘uncaught’ because of .catch().

You can also use the .catch() method in a chain of promises. It catches the first error it encounters in the chain.

For instance, refactoring the chain of promises following the fetchResource() function from the example of the previous section. This how you can stop error callback repetition in your code.

fetchResource(url)

.then(handleResult)

.then(handleNewResult)

.then(handleAnotherResult)

.catch(failureCallback);

You can also use .catch() to check for errors in a group of promises before proceeding with further asynchronous operations.

fetchResource(url)

.then(handleResult)

.then(handleNewResult)

.catch(failureCallback)

// Check for Errors in the above group of promises before proceeding

.then(handleAnotherResult);

The .catch() method addresses any errors in a promise without requiring the nesting of error callback functions.

To chain an asynchronous operation to a promise regardless of if the promise is resolved or not, use the .finally() method. The .then() method is how you handle the results of a promise writing individual conditions for both resolved and rejected. .catch() runs only when there is an error. But sometimes you might want an operation to run no matter what happens to earlier promises.

Using finally() helps prevent possible code repetition in .then() and .catch(). It is for operations you must run whether there is an error or not.

fetchResource(url)

.then(handleResult)

.then(handleNewResult)

.finally(onFinallyHandle);

The finally() method has a few use cases in real-world applications. It is important if you want to perform cleanup operations for activities the promise initiated. Another use case—on Front-End Web Applications—is making user interface updates like stopping a loading spinner.

## How to Handle Many Promises at Once

It is possible to run more than one promise at a time. All the examples you have seen so far are for promises that run one after the other.

In the previous examples, promises run similarly to synchronous code in the sense that they wait for the previous one to be resolved or rejected. But you could have multiple promises that run in parallel.

Here are the available methods that can help us achieve this:

* Promise.all()
* Promise.race()
* Promise.any()
* Promise.allSettled()

In this section of the article, we'll review these methods.

### The Promise.all() method

Promise.all() accepts an array of promises as an argument but returns a single promise as the output. The single promise it returns resolves with an array of values if all the promises in the input array are fulfilled. The array Promise.all() resolves with will contain the resolve values of individual promises in the input array.

const promise1 = Promise.resolve(`First Promise's Value`);

const promise2 = new Promise((resolve) =>

setTimeout(resolve, 3000, `Second Promise's Value`)

);

const promise3 = new Promise((resolve) =>

setTimeout(resolve, 2000, `Third Promise's Value`)

);

Promise.all([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<fulfilled>: Array(3)}\*

Promise.all([promise1, promise2, promise3]).then((values) => {

values.forEach((value) => console.log(value));

});

// Output on the console

// First Promise's Value

// Second Promise's Value

// Third Promise's Value

If at least one promise in the input array does not resolve, Promise.all() will return a rejected promise with a reason. The reason for the rejection will be the same as that of the first rejected promise in the input array.

const promise1 = Promise.resolve(`First Promise's Value`);

const promise2 = new Promise((resolve, reject) =>

setTimeout(reject, 3000, `First reason for rejection`)

);

const promise3 = new Promise((resolve, reject) =>

setTimeout(reject, 2000, `Second reason for rejection`)

);

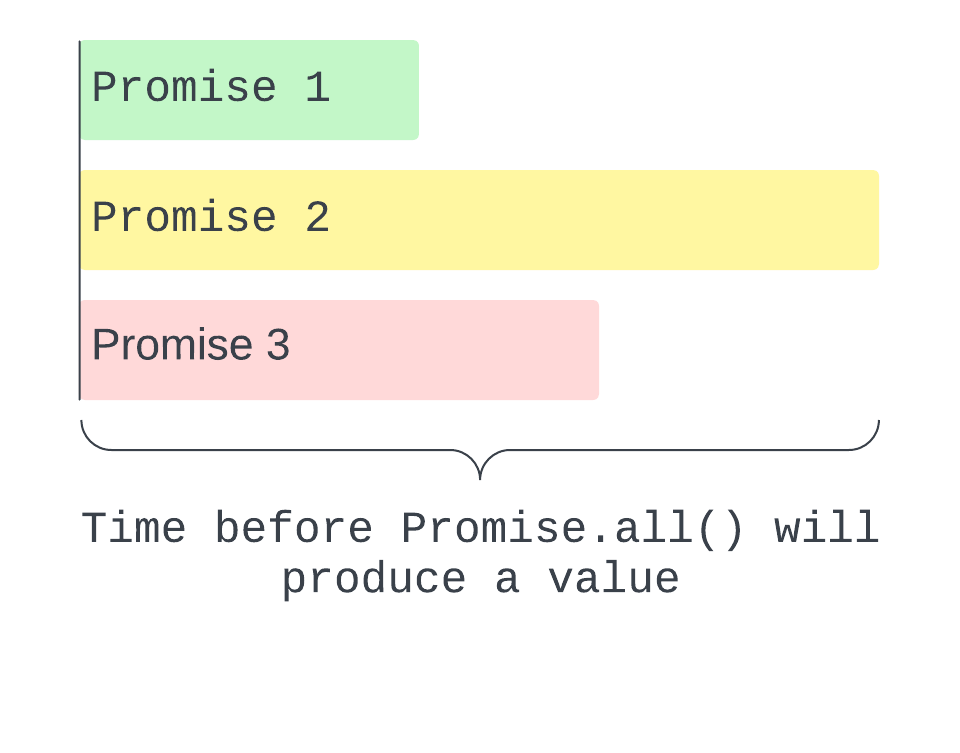
Promise.all([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<rejected>: "First reason for rejection"}\*

Promise.all() will run all the input promises before it returns a value. But it does not run the promises one after the other–instead it runs them at the same time.

This is why the total time it would take Promise.all() to return a value is roughly the time it would take the longest promise in the array to finish.



Despite that, it has to finish running all the promises before it returns anything.

### The Promise.race() method

Promise.race() accepts an array of promises as an argument and returns a single promise as an output. The single promise it returns is the fastest promise to finish running—resolved or not. This means Promise.race() will return the promise with the shortest execution time in an array of promises.

const promise1 = new Promise((resolve) =>

setTimeout(resolve, 3000, `First Promise's Value`)

);

const promise2 = new Promise((resolve) =>

setTimeout(resolve, 2000, `Second Promise's Value`)

);

const promise3 = Promise.resolve(`Third Promise's Value`);

Promise.race([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<fulfilled>: "Third Promise's Value"}\*

In the example above, because promise3 is a promise that resolves on being created, Promise.race() returns it as the fastest. Just like other Promise methods the article discusses in this section, it runs the promises in parallel and not one after the other.

If the promise with the shortest execution time happens to be rejected with a reason, Promise.race() returns a rejected promise and the reason why the fastest promise was rejected.

const promise1 = Promise.reject(`Reason for rejection`);

const promise2 = new Promise((resolve) =>

setTimeout(resolve, 3000, `First resolved Promise`)

);

const promise3 = new Promise((resolve) =>

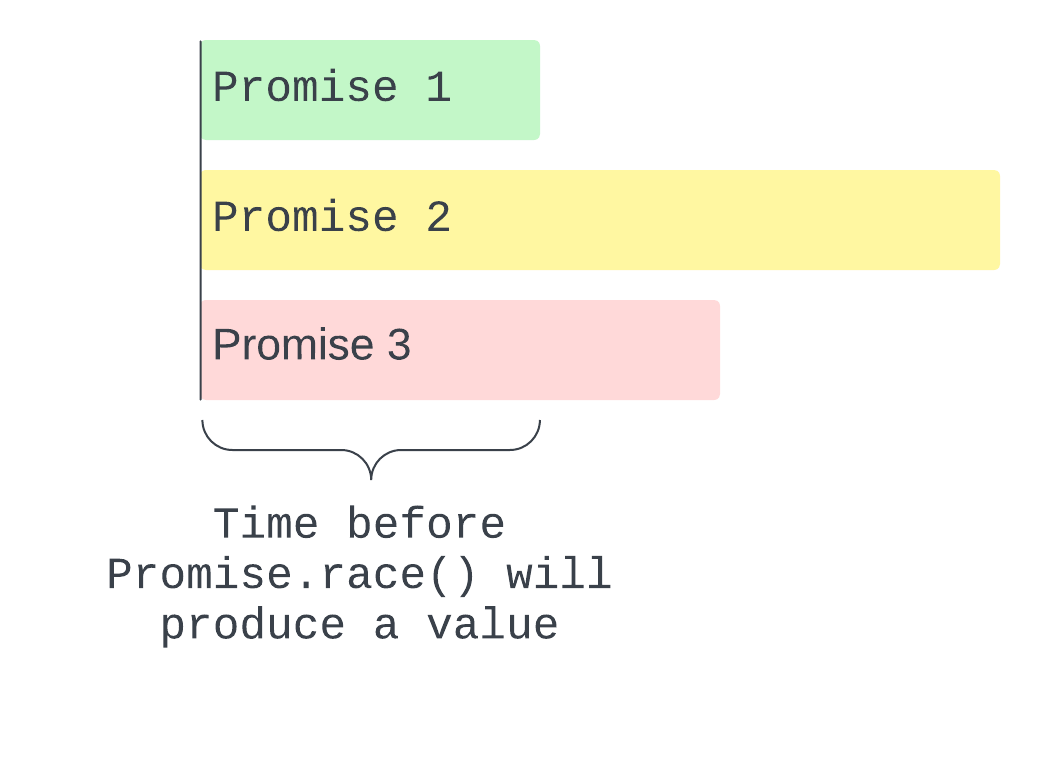
setTimeout(resolve, 2000, `Second resolved Promise`)

);

Promise.race([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<rejected>: "Reason for rejection"}\*



Promise.race() is useful for running a list of asynchronous operations but only needing the result of the fastest executed operation.

### The Promise.any() method

Promise.any() accepts an array of Promises as an argument but returns a single Promise as the output. The single promise it returns is the first resolved promise in the input array. This method waits for any promise in the array to be resolved and would immediately return it as the output.

const promise1 = new Promise((resolve) =>

setTimeout(resolve, 3000, `First Promise's Value`)

);

const promise2 = new Promise((resolve) =>

setTimeout(resolve, 2000, `Second Promise's Value`)

);

const promise3 = Promise.reject(`Third Promise's Value`);

Promise.any([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<fulfilled>: "Second Promise's Value"}\*

From the above example, promise1 will resolve after 3 seconds, promise2 will resolve after 2 seconds, and promise3 immediately rejects. Because Promise.any() is looking for the first successful promise, it returns promise2. promise1 is a little bit late and so it's left behind.

If none of the promises in the array are resolved, Promise.any() returns a rejected promise. This rejected promise contains a JavaScript array of reasons, where each reason corresponds with that of a promise from the input array.

const promise1 = new Promise((resolve, reject) =>

setTimeout(reject, 3000, `First rejection reason`)

);

const promise2 = new Promise((resolve, reject) =>

setTimeout(reject, 2000, `Second rejection reason`)

);

const promise3 = Promise.reject(`Third rejection reason`);

Promise.any([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<rejected>: Aggregate Error: All Promises were rejected}\*

Promise.any([promise1, promise2, promise3]).catch(({ errors }) =>

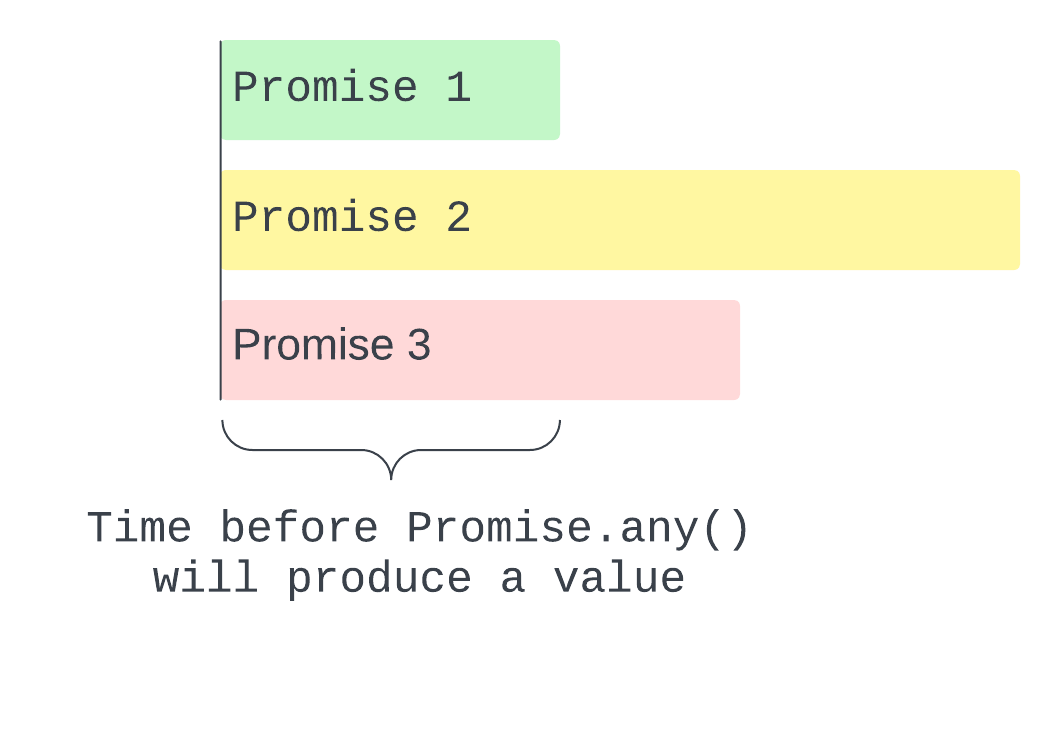
console.log(errors)

);

// Output on the console

// \*(3) ["First\* rejection reason\*", "Second\* rejection reason\*", "Third\* rejection reason\*"]\*

This method is useful for asynchronous operations where the fastest successful promise is all you need. Promise.any() and Promise.race() are similar, except that Promise.any() will return the fastest promise to complete and be resolved, while Promise.race() will return the fastest promise to complete and does not care if it is resolved or not.



### The Promise.allSettled() method

Promise.allSettled() became a feature of JavaScript promises with the release of [ES2020](https://262.ecma-international.org/11.0/). It handles promises in parallel just like the other promise methods the article discusses in this section.

Promise.allSettled() helps to write more efficient asynchronous code as it shows the outcome of all the promises in the array regardless of the status—resolved or rejected.

Promise.allSettled() accepts an array of promises as an argument and returns a single promise as the output.

The single promise it returns will always resolve or enter the state ‘fulfilled’ after all the input promises are settled. It does not care if any individual promise in the input array rejected. The array Promise.all() resolves with will contain the resolve values or rejection reasons of promises in the input array.

const promise1 = new Promise((resolve) =>

setTimeout(resolve, 3000, `First Promise's Value`)

);

const promise2 = new Promise((resolve) =>

setTimeout(resolve, 2000, `Second Promise's Value`)

);

const promise3 = Promise.reject(`Third Promise's Value`);

Promise.allSettled([promise1, promise2, promise3]);

// Output on the console

// \*Promise {<fulfilled>: Array(3)}\*

Promise.allSettled([promise1, promise2, promise3]).then(console.log);

// Output on the console

/\*

(3) [{…}, {…}, {…}]

0: {status: 'fulfilled', value: "First Promise's Value"}

1: {status: 'fulfilled', value: "Second Promise's Value"}

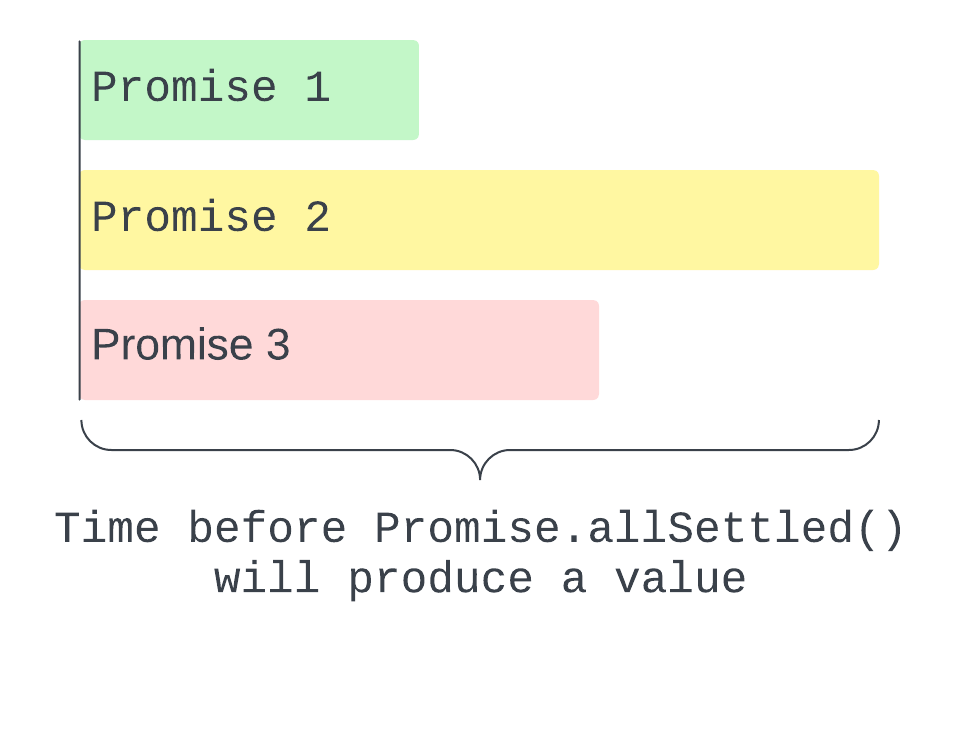
2: {status: 'rejected', reason: "Third Promise's Value"}

\*/

From the example above, you can see that even though promise3 rejects on creation, Promise.allSettled() still returned a ‘fulfilled’ promise. It does this even if all the promises in the input array reject.

Promise.allSettled() is similar to Promise.all() in that all their input promises must settle before the promise they return has a settled state—fulfilled or rejected.

The difference is Promise.all() can only be successful if all the promises in the input are resolved, while Promise.allSettled() does not care about the status of the input promises.



Using this method will give you an overview of how all your promises did, the ones that were resolved and the ones that were rejected. It gives complete information on all the promises you pass into it and allows you to examine them independently—the outcome of one does not affect the state of the promise the method returns.

## What is the Async/Await Syntax?

Async/await syntax became a feature of JavaScript with the release of [ES8(ES2017)](https://262.ecma-international.org/8.0/). It is built on top of promises, and you can see it as an alternative syntax to promises.

async/await eliminates the chaining that is common with the promises syntax, and ends up making asynchronous code look a lot more synchronous.

Promises are an excellent way to avoid the previously discussed ‘Callback Pyramid of Doom’, but async/await takes asynchronous code further. With async/await, code is easier to follow and maintain. It came about as a way to improve code readability for asynchronous operations. It is the modern way of using promises.

### How to Create an Async Function in JavaScript

async is a JavaScript keyword used to create a function. The function this keyword helps create will always return a promise. To use it, place async before the function keyword when declaring the function.

async function example() {

// Return a value

}

example()

// Output on the console

// \*Promise {<fulfilled>: undefined}\*

From the code example, you can see that the function returns a promise with a value undefined. This is because anything the async function returns will be the resolved value of the resulting promise. In this case, the function does not return anything, hence undefined.

async function example() {

return "Feels good to be an async function";

}

example();

// Output on the console

// \*Promise {<fulfilled>: "Feels good to be an async function"}\*

In the above example, the function returns a string, which becomes the resolved value of the resulting promise. That is the way to create an async function.

### How to Use the Await Keyword

To use the await keyword, place it before a promise. It is an indicator for the async function to pause execution until that promise is settled.

It is similar to the .then() method which makes sure a promise is ‘fulfilled’ or ‘rejected’ before it continues. Note that you can only use the await keyword inside an async function.

Instead of chaining promises with .then() as the article earlier teaches, you can repeatedly await the asynchronous operations making your code cleaner and easier to read.

const timerPromise = (message) =>

new Promise((resolve) => setTimeout(resolve, 3000, message));

async function asyncFunc() {

const result = await timerPromise("promise finished!");

console.log(result);

}

// Output on the Console after 3 seconds

// promise finished!

Using the await keyword before a promise will produce the resolved value of that promise. It is evident from the line const result = await promise('promise finished!') where  result becomes a string and not a new promise. This is different from .then() which always returns a new promise.

With await, you can break up any chain of promises, and grab their resolve values. The following example uses the fetch() function—which is a promise—to show eliminating chaining with async/await.

// With chaining

fetch("https://jsonplaceholder.typicode.com/users")

.then((response) => response.json())

.then((result) => console.log(result));

// Output on the console

// Array(10) [...]

// Without chaining

async function fetchResource(url) {

const response = await fetch(url);

const result = await response.json();

console.log(result);

}

fetchResource("https://jsonplaceholder.typicode.com/users");

// Output on the console

// Array(10) [...]

In the end, it boils down to preference and choice. If you prefer the chaining syntax, then go for it. If you prefer your code to look synchronous and want to use async/await, then that is fine too.

You can also use both syntaxes together, chaining promises inside an async function. It all depends on what you want to achieve and the style you prefer.

### How to Handle Errors in Async/Await

Just like with the normal promise syntax, you can catch errors properly using async/await. Properly handling errors in async calls is extremely important to track bugs. Use try/catch blocks to do this.

try is a JavaScript keyword that wraps a block of code. As that block of code runs, try checks for errors. No error can escape a try block. Use try inside an async function.

The first error inside the try block stops the other instructions in that block from executing, try then passes the error value to the catch block. The catch block is similar to .catch() in promises. Just like the promise method, it is a function of an error.

async function fetchResource(url) {

try {

const response = await fetch(url);

const result = await response.json();

console.log(result);

} catch (error) {

console.error(error);

}

}

In this example, the catch keyword has an error, which logs to the console. A settled promise with an uncaught error results in a rejected promise. Make sure you wrap your code in try/catch blocks to have more control over failures and faults in your program.

Also, just like the .finally() method for promises, you can use a finally block inside an async function. Braces that follow this keyword wrap around a block of code that would run regardless of if there is an error or not.

async function fetchResource(url) {

try {

const response = await fetch(url);

const result = await response.json();

console.log(result);

} catch (error) {

console.error(error);

} finally {

console.log("Operation finished!");

}

}

The use of the finally block is similar to the use of the .finally() method. This just proves that using an async function is a recent way to work with promises.

## What is a Job Queue?

The Job Queue—also known as the Microtask Queue—is an important concept to be aware of in JavaScript. It was not originally a component of the JavaScript runtime, but the need for it came when promises became a part of JavaScript.

Consider the following code sample:

Promise.resolve("This is a resolved value").then(console.log);

setTimeout(console.log, 0, "This is a value after the timeout");

console.log("This is a normal log");

Here the first line is a promise that automatically resolves, then logs the value on the console. The second line is a timeout set to 0 milliseconds which means it is supposed to be instant. The timeout takes in a callback function that logs a value to the console. The third line is a normal console log.

When you run the program, can you guess the order in which these logs would appear? Let's find out.

// Output on the console

/\*

This is a normal log

This is a resolved value

--

undefined

--

This is a value after the timeout

\*/

This is an interesting output. The first log is from console.log. It is not so confusing because console.log() is not an async operation. The JavaScript engine will actively run every synchronous instruction immediately after the program starts.

The second line might be a little puzzling. It logs the resolved value of the promise. Why does the output from the promise come next? Well, the simple answer is that a promise is faster than any other async implementation in JavaScript. But that is not the full story.

In the JavaScript runtime, the event loop handles async operations. It can only call the callback functions of async instructions when the call stack is empty. Resolving a promise is an asynchronous operation, and it is understandable that it comes after a normal log. But why does it come before the setTimeout() instruction?

The JavaScript Runtime actually has these two queues—the Callback (or Macrotask) Queue and the Job (or Microtask) Queue. Shortly before the event loop starts calling the functions in the Callback Queue, it calls all the instructions on the Job Queue. The callback of a promise stays in the Job Queue so the event loop calls it first. This is why promises return values faster than any other async implementation.

The Job Queue is useful for some other instructions apart from promises. However, that is beyond the scope of this material. If you are curious, then you can [read more about the Job Queue here](https://blog.greenroots.info/task-queue-and-job-queue-deep-dive-into-javascript-event-loop-model).

A program returns immediately after taking care of the Job Queue. From the above code example, it returns with undefined. After that, the event loop moves over to the Callback Queue and executes the instructions there.

# JavaScript – AJAX

<https://www.tutorialrepublic.com/javascript-tutorial/javascript-ajax.php>

<https://www.geeksforgeeks.org/how-to-make-ajax-call-from-javascript/>

## What is Ajax?

Ajax stands for **A**synchronous **J**avascript **A**nd **X**ml. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the browser's built-in XMLHttpRequest (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

**Note:** Ajax is not a new technology, in fact, Ajax is not even really a technology at all. Ajax is just a term to describe the process of exchanging data from a web server asynchronously through JavaScript, without refreshing the page.

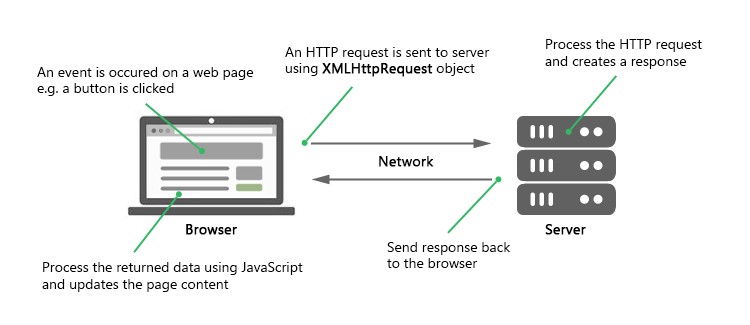
**Tip:** Don't get confused by the term **X** (i.e. **XML**) in AJAX. It is only there for historical reasons. Other data exchange format such as JSON, HTML, or plain text can be used instead of XML.

### Understanding How Ajax Works

To perform Ajax communication JavaScript uses a special object built into the browser—an XMLHttpRequest (XHR) object—to make HTTP requests to the server and receive data in response.

All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the XMLHttpRequest object.

The following illustrations demonstrate how Ajax communication works:



Since Ajax requests are usually asynchronous, execution of the script continues as soon as the Ajax request is sent, i.e. the browser will not halt the script execution until the server response comes back.

In the following section we'll discuss each step involved in this process one by one:

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an XMLHttpRequest object, as shown below:

var request = new XMLHttpRequest();

Now, the next step in sending the request to the server is to instantiating the newly-created request object using the open() method of the XMLHttpRequest object.

The open() method typically accepts two parameters— the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");

**Tip:** The file can be of any kind, like .txt or .xml, or server-side scripting files, like .php or .asp, which can perform some actions on the server (e.g. inserting or reading data from database) before sending the response back to the client.

And finally send the request to the server using the send() method of the XMLHttpRequest object.

request.send(); -Or- request.send(*body*);

**Note:** The send() method accepts an optional body parameter which allow us to specify the request's body. This is primarily used for HTTP POST requests, since the HTTP GET request doesn't have a request body, just request headers.

The GET method is generally used to send small amount of data to the server. Whereas, the POST method is used to send large amount of data, such as form data.

In GET method, the data is sent as URL parameters. But, in POST method, the data is sent to the server as a part of the HTTP request body. Data sent through POST method will not visible in the URL.

See the chapter on [HTTP GET vs. POST](https://www.tutorialrepublic.com/php-tutorial/php-get-and-post.php) for a detailed comparison of these two methods.

In the following section we'll take a closer look at how Ajax requests actually works.

### Performing an Ajax GET Request

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change in database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript (**Gives CORS error**).

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax GET Demo</title>

<script>

function displayFullName() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("GET", "https://www.tutorialrepublic.com/examples/php/greet.php?fname=John&lname=Clark");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is compete and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Sending the request to the server

request.send();

}

</script>

</head>

<body>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

<button type="button" onclick="displayFullName()">Display Full Name</button>

</body>

</html>

**Another example:**

<script>

    function run() {

        // Creating Our XMLHttpRequest object

        var xhr = new XMLHttpRequest();

        // Making our connection

        var url = '<https://jsonplaceholder.typicode.com/todos/1>';

        xhr.open("GET", url, true);

        // function execute after request is successful

        xhr.onreadystatechange = function () {

            if (this.readyState == 4 && this.status == 200) {

                console.log(this.responseText);

            }

        }

        // Sending our request

        xhr.send();

    }

    run();

</script>

**Output:**

"{

"userId": 1,

"id": 1,

"title": "delectus aut autem",

"completed": false

}"

When the request is asynchronous, the send() method returns immediately after sending the request. Therefore you must check where the response currently stands in its lifecycle before processing it using the readyState property of the XMLHttpRequest object.

The readyState is an integer that specifies the status of an HTTP request. Also, the function assigned to the onreadystatechange event handler called every time the readyState property changes. The possible values of the readyState property are summarized below.

|  |  |  |
| --- | --- | --- |
| **Value** | **State** | **Description** |
| 0 | UNSENT | An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized). |
| 1 | OPENED | The open() method has been called (i.e. server connection established). |
| 2 | HEADERS\_RECEIVED | The send() method has been called (i.e. server has received the request). |
| 3 | LOADING | The server is processing the request. |
| 4 | DONE | The request has been processed and the response is ready. |

**Note:** Theoretically, the readystatechange event should be triggered every time the readyState property changes. But, most browsers do not fire this event when readyState changes to 0 or 1. However, all browsers fire this event when readyState changes to 4 .

The status property returns the numerical HTTP status code of the XMLHttpRequest's response. Some of the common HTTP status codes are listed below:

* 200 — OK. The server successfully processed the request.
* 404 — Not Found. The server can't find the requested page.
* 503 — Service Unavailable. The server is temporarily unavailable.

Please check out the [HTTP status codes](https://www.tutorialrepublic.com/html-reference/http-status-codes.php) reference for a complete list of response codes.

Here's the code from our "greet.php" file that simply creates the full name of a person by joining their first name and last name and outputs a greeting message.

<?php

if(isset($\_GET["fname"]) && isset($\_GET["lname"])) {

$fname = htmlspecialchars($\_GET["fname"]);

$lname = htmlspecialchars($\_GET["lname"]);

// Creating full name by joining first and last name

$fullname = $fname . " " . $lname;

// Displaying a welcome message

echo "Hello, $fullname! Welcome to our website.";

} else {

echo "Hi there! Welcome to our website.";

}

?>

### Performing an Ajax POST Request

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax POST Demo</title>

<script>

function postComment() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("POST", "/examples/php/confirmation.php");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is compete and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Retrieving the form data

var myForm = document.getElementById("myForm");

var formData = new FormData(myForm);

// Sending the request to the server

request.send(formData);

}

</script>

</head>

<body>

<form id="myForm">

<label>Name:</label>

<div><input type="text" name="name"></div>

<br>

<label>Comment:</label>

<div><textarea name="comment"></textarea></div>

<p><button type="button" onclick="postComment()">Post Comment</button></p>

</form>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

</body>

</html>

If you are not using the FormData object to send form data, for example, if you're sending the form data to the server in the query string format, i.e. request.send(key1=value1&key2=value2) then you need to [explicitly set the request header](https://www.tutorialrepublic.com/codelab.php?topic=javascript&file=set-request-header) using setRequestHeader() method, like this:

request.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

The setRequestHeader() method, must be called after calling open(), but before calling send().

Some common request headers are: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml, text/plain, text/html, and so on.

**Note:** The FormData object provides an easy way to construct a set of key/value pairs representing form fields and their values which can be sent using XMLHttpRequest.send() method. The transmitted data is in the same format that the form's submit() method would use to send the data if the form's encoding type were set to multipart/form-data.

Here's the code of our "confirmation.php" file that simply outputs the values submitted by the user.

<?php

if($\_SERVER["REQUEST\_METHOD"] == "POST") {

$name = htmlspecialchars(trim($\_POST["name"]));

$comment = htmlspecialchars(trim($\_POST["comment"]));

// Check if form fields values are empty

if(!empty($name) && !empty($comment)) {

echo "<p>Hi, <b>$name</b>. Your comment has been received successfully.<p>";

echo "<p>Here's the comment that you've entered: <b>$comment</b></p>";

} else {

echo "<p>Please fill all the fields in the form!</p>";

}

} else {

echo "<p>Something went wrong. Please try again.</p>";

}

?>

For security reasons, browsers do not allow you to make cross-domain Ajax requests. This means you can only make Ajax requests to URLs from the same domain as the original page, for example, if your application is running on the domain "mysite.com", you cannot make Ajax request to "othersite.com" or any other domain. This is commonly known as same origin policy.

However, you can load images, style sheets, JS files, and other resources from any domain.

## Fetch

The [**fetch()**](https://www.geeksforgeeks.org/fetch-api/)API which is used to make XMLHttpRequest with the server. Because of its flexible structure, it is easy to use. This API makes a request to the server and gets the result as a [promise](https://www.geeksforgeeks.org/javascript-promises/) which is resolved to the string.

**Syntax:**

fetch(url, {config}).then().catch();

**Parameter:**It takes URL and config of request as parameters.

We will configure the data required and make the request to the server. Since it is a resolved promise we use [**then()**](https://www.geeksforgeeks.org/why-we-use-then-method-in-javascript/)function and **catch()** function to create output for the result. In response, we get the string that we print.

**Example:**

|  |
| --- |
| <script>        // Url for the request  **var** url = '<https://jsonplaceholder.typicode.com/todos/1>';        // Making our request      fetch(url, { method: 'GET' })          .then(Result => Result.json())          .then(string => {                // Printing our response              console.log(string);                // Printing our field of our response              console.log(`Title of our response :  ${string.title}`);          })          .**catch**(errorMsg => { console.log(errorMsg); });  </script> |

**Output:**

{ userId:1 ,id:1 ,title : "delectus aut autem" ,completed : false

\_\_proto\_\_:Object }

Title of our response : delectus aut autem

# JavaScript – JSON

<https://www.w3schools.com/js/js_json.asp>

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

* JSON stands for **J**ava**S**cript **O**bject **N**otation
* JSON is a lightweight data interchange format
* JSON is language independent **\***
* JSON is "self-describing" and easy to understand

\* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

**JSON Example**

This JSON syntax defines an employees object: an array of 3 employee records (objects):

{  
"employees":[  
  {"firstName":"John", "lastName":"Doe"},  
  {"firstName":"Anna", "lastName":"Smith"},  
  {"firstName":"Peter", "lastName":"Jones"}  
]  
}

## JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

"firstName":"John"

JSON names require double quotes. JavaScript names do not.

## JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

{"firstName":"John", "lastName":"Doe"}

## JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

"employees":[  
  {"firstName":"John", "lastName":"Doe"},  
  {"firstName":"Anna", "lastName":"Smith"},  
  {"firstName":"Peter", "lastName":"Jones"}  
]

In the example above, the object "employees" is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

## Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

let text = '{ "employees" : [' +

'{ "firstName":"John" , "lastName":"Doe" },' +

'{ "firstName":"Anna" , "lastName":"Smith" },' +

'{ "firstName":"Peter" , "lastName":"Jones" } ]}';

Then, use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

const obj = JSON.parse(text);

Finally, use the new JavaScript object in your page:

<!DOCTYPE html>

<html>

<body>

<h2>Create Object from JSON String</h2>

<p id="demo"></p>

<script>

let text = '{"employees":[' +

'{"firstName":"John","lastName":"Doe" },' +

'{"firstName":"Anna","lastName":"Smith" },' +

'{"firstName":"Peter","lastName":"Jones" }]}';

const obj = JSON.parse(text);

document.getElementById("demo").innerHTML =

obj.employees[1].firstName + " " + obj.employees[1].lastName;

</script>

</body>

</html>

# HTTP – CORS

<https://portswigger.net/web-security/cors>

<https://web.dev/cross-origin-resource-sharing/>

## What is CORS (cross-origin resource sharing)?

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy ([SOP](https://portswigger.net/web-security/cors/same-origin-policy)). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as [cross-site request forgery](https://portswigger.net/web-security/csrf) (CSRF).

## Same-origin policy

The same-origin policy is a restrictive cross-origin specification that limits the ability for a website to interact with resources outside of the source domain. The same-origin policy was defined many years ago in response to potentially malicious cross-domain interactions, such as one website stealing private data from another. It generally allows a domain to issue requests to other domains, but not to access the responses.

The browser's same-origin policy blocks reading a resource from a different origin. This mechanism stops a malicious site from reading another site's data, but it also prevents legitimate uses. What if you wanted to get weather data from another country?

In a modern web application, an application often wants to get resources from a different origin. For example, you want to retrieve JSON data from a different domain or load images from another site into a <canvas> element.

In other words, there are **public resources** that should be available for anyone to read, but the same-origin policy blocks that. Developers have used work-arounds such as [JSONP](https://stackoverflow.com/questions/2067472/what-is-jsonp-all-about), but **Cross-Origin Resource Sharing (CORS)** fixes this in a standard way.

Enabling **CORS** lets the server tell the browser it's permitted to use an additional origin.

A browser and a server can exchange data over the network using the **Hypertext Transfer Protocol** (HTTP). HTTP defines the communication rules between the requester and the responder, including what information is needed to get a resource.

The HTTP header is used to negotiate the type of message exchange between the client and the server and is used to determine access. Both the browser's request and the server's response message are divided into two parts: **header** and **body**:

### header

Information about the message such as the type of message or the encoding of the message. A header can include a [variety of information](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) expressed as key-value pairs. The request header and response header contain different information.

**Sample Request header**

Accept: text/html

Cookie: Version=1

The above is equivalent to saying "I want to receive HTML in response. Here is a cookie I have."

Sample Response header

Content-Encoding: gzip

Cache-Control: no-store

The above is equivalent to saying "Data is encoded with gzip. Do not cache this please."

### body

The message itself. This could be plain text, an image binary, JSON, HTML, and so on.

## How does CORS work?

Remember, the same-origin policy tells the browser to block cross-origin requests. When you want to get a public resource from a different origin, the resource-providing server needs to tell the browser "This origin where the request is coming from can access my resource". The browser remembers that and allows cross-origin resource sharing.

**Step 1: client (browser) request**

When the browser is making a cross-origin request, the browser adds an Origin header with the current origin (scheme, host, and port).

**Step 2: server response**

On the server side, when a server sees this header, and wants to allow access, it needs to add an Access-Control-Allow-Origin header to the response specifying the requesting origin (or \* to allow any origin.)

**Step 3: browser receives response**

When the browser sees this response with an appropriate Access-Control-Allow-Origin header, the browser allows the response data to be shared with the client site.

## See CORS in action

Here is a tiny web server using Express.

const express = require('express');

const app = express();

// No CORS Headder set

app.get('/', function(request, response) {

response.sendFile(\_\_dirname + '/message.json');

});

// CORS header `Access-Control-Allow-Origin` set to accept all

app.get('/allow-cors', function(request, response) {

response.set('Access-Control-Allow-Origin', '\*');

response.sendFile(\_\_dirname + '/message.json');

});

// listen for requests :)

const listener = app.listen(process.env.PORT, function() {

console.log('Your app is listening on port ' + listener.address().port);

});

The first endpoint (line 8) does not have any response header set, it just sends a file in response.

* Press `Control+Shift+J` (or `Command+Option+J` on Mac) to open DevTools.
* Press `Control+Shift+J` (or `Command+Option+J` on Mac) to open DevTools.
* Click the **Console** tab.
* Try the following command:

fetch('https://cors-demo.glitch.me/', {mode:'cors'})

**You should see an error saying:**

request has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header  
is present on the requested resource.

The second endpoint (line 13) sends the same file in response but adds Access-Control-Allow-Origin: \* in the header. From the console, try

fetch('https://cors-demo.glitch.me/allow-cors', {mode:'cors'})

This time, your request should not be blocked.