

TutorialThe JavaScript languageAn introduction{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"{"@type":"ListItem","position":3,"name":"An introduction","item":"https://javascript.info/getting-started"}]}August 8, 2022An Introduction to JavaScriptLet's see what's so special about JavaScript, what we can achieve with it, and what other technologies play well with it. 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, 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. The browser has an embedded engine sometimes called a “JavaScript virtual machine”. Different engines have different “codenames”. For example:

V8 – in Chrome, Opera and Edge.

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.

The engine (embedded if it's a browser) reads (“parses”) the script. Then it converts (“compiles”) the script to machine code. 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? 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 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 and COMET 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? 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.

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? 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.

Languages "over" JavaScript The syntax of JavaScript does not suit everyone's needs.

Different people want different features.

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

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

Modern tools make the transpilation very fast and transparent, actually allowing developers to code in another language and auto-converting it "under the hood".

Examples of such languages:

CoffeeScript is "syntactic sugar" for JavaScript. It introduces shorter syntax, allowing us to write clearer and more precise code. Usually, Ruby devs like it.

TypeScript is concentrated on adding "strict data typing" to simplify the development and support of complex systems. It is developed by Microsoft.

Flow also adds data typing, but in a different way. Developed by Facebook.

Dart is a standalone language that has its own engine that runs in non-browser environments (like mobile apps), but also can be transpiled to JavaScript. Developed by Google.

Brython is a Python transpiler to JavaScript that enables the writing of applications in pure Python without JavaScript.

Kotlin is a modern, concise and safe programming language that can target the browser or Node.

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

Summary

JavaScript was initially created as a browser-only language, but it is now used in many other environments as well.

Today, JavaScript has a unique position as the most widely-adopted browser language, fully integrated with HTML/CSS.

There are many languages that get “transpiled” to JavaScript and provide certain features. It is recommended to take a look at them, at least briefly, after mastering JavaScript.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines –

wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, {“url”:“https://javascript.info/intro”,“identifier”:“Vintro”}); };var DisqusShortname = “javascriptinfo”;var DisqusEnabled = true;

TutorialThe JavaScript languageAn introduction{“@context”:“https://schema.org”,“@type”:“BreadcrumbList”,“itemListElement”:

[{“@type”:“ListItem”,“position”:1,“name”:“Tutorial”,“item”:“https://javascript.info/”},

{“@type”:“ListItem”,“position”:2,“name”:“The JavaScript language”,“item”:“https://

javascript.info/js”},{“@type”:“ListItem”,“position”:3,“name”:“An

introduction”,“item”:“https://javascript.info/getting-started”}]]July 10, 2022Manuals and

specificationsThis book is a tutorial. It aims to help you gradually learn the language.

But once you’re familiar with the basics, you’ll need other resources.

SpecificationThe ECMA-262 specification contains the most in-depth, detailed and formalized information about JavaScript. It defines the language.

But being that formalized, it’s difficult to understand at first. So if you need the most trustworthy source of information about the language details, the specification is the right place. But it’s not for everyday use.

A new specification version is released every year. Between these releases, the latest specification draft is at <https://tc39.es/ecma262/>.

To read about new bleeding-edge features, including those that are “almost standard” (so-called “stage 3”), see proposals at <https://github.com/tc39/proposals>.

Also, if you’re developing for the browser, then there are other specifications covered in the second part of the tutorial.

Manuals

MDN (Mozilla) JavaScript Reference is the main manual with examples and other information. It’s great to get in-depth information about individual language functions, methods etc.

You can find it at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>.

Although, it’s often best to use an internet search instead. Just use “MDN [term]” in the query, e.g. <https://google.com/search?q=MDN+parseInt> to search for the parseInt function.

Compatibility tables JavaScript is a developing language, new features get added regularly.

To see their support among browser-based and other engines, see:

<https://caniuse.com> – per-feature tables of support, e.g. to see which engines support modern cryptography functions: <https://caniuse.com/#feat=cryptography>.

<https://kangax.github.io/compat-table> – a table with language features and engines that support those or don't support.

All these resources are useful in real-life development, as they contain valuable information about language details, their support, etc.

Please remember them (or this page) for the cases when you need in-depth information about a particular feature.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/manuals-specifications", "identifier": "manuals-specifications" }); };  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;
```

TutorialThe JavaScript languageAn introduction{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement":

```
[{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"},  
{"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"},  
{"@type":"ListItem", "position":3, "name":"An introduction", "item":"https://javascript.info/getting-started"}]]
```

October 18, 2022Code

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

There are two main types of code editors: IDEs and lightweight editors. Many people

use one tool of each type.

IDEThe term IDE (Integrated Development Environment) refers to a powerful editor with many features that usually operates on a “whole project.” As the name suggests, it's not just an editor, but a full-scale “development environment.”

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

If you haven't selected an IDE yet, consider the following options:

Visual Studio Code (cross-platform, free).

WebStorm (cross-platform, paid).

For Windows, there's also “Visual Studio”, not to be confused with “Visual Studio Code”.

“Visual Studio” is a paid and mighty Windows-only editor, well-suited for the .NET platform. It's also good at JavaScript. There's also a free version Visual Studio Community.

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

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

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

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

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

There are many options, for instance:

Sublime Text (cross-platform, shareware).

Notepad++ (Windows, free).

Vim and Emacs are also cool if you know how to use them.

Let's not argueThe editors in the lists above are those that either I or my friends whom I consider good developers have been using for a long time and are happy with.

There are other great editors in our big world. Please choose the one you like the most.

The choice of an editor, like any other tool, is individual and depends on your projects, habits, and personal preferences.

The author's personal opinion:

I'd use Visual Studio Code if I develop mostly frontend.

Otherwise, if it's mostly another language/platform and partially frontend, then consider other editors, such as XCode (Mac), Visual Studio (Windows) or JetBrains family (Webstorm, PhpStorm, RubyMine etc, depending on the language).

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/code-editors", "identifier": "Vcode-editors" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageAn introduction{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "An introduction", "item": "https://javascript.info/getting-started"}]}April 5, 2021Developer
consoleCode is prone to errors. You will quite likely make errors... Oh, what am I talking about? You are absolutely going to make errors, at least if you're a human, not a robot. But in the browser, users don't see errors by default. So, if something goes wrong in the

script, we won't see what's broken and can't fix it.

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

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

Developer tools are potent; they have many features. To start, we'll learn how to open them, look at errors, and run JavaScript commands.

Google ChromeOpen the page `bug.html`.

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

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

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

It looks somewhat like this:

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

Here we can see the red-colored error message. In this case, the script contains an unknown "lalala" command.

On the right, there is a clickable link to the source `bug.html:12` with the line number where the error has occurred.

Below the error message, there is a blue > symbol. It marks a "command line" where we can type JavaScript commands. Press Enter to run them.

Now we can see errors, and that's enough for a start. We'll come back to developer tools later and cover debugging more in-depth in the chapter Debugging in the browser.

Multi-line input

Usually, when we put a line of code into the console, and then press Enter, it executes.

To insert multiple lines, press `Shift+Enter`. This way one can enter long fragments of JavaScript code.

Firefox, Edge, and othersMost other browsers use F12 to open developer tools.

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

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

Open Preferences and go to the "Advanced" pane. There's a checkbox at the bottom:

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

Summary

Developer tools allow us to see errors, run commands, examine variables, and much more.

They can be opened with F12 for most browsers on Windows. Chrome for Mac needs Cmd+Opt+J, Safari: Cmd+Opt+C (need to enable first).

Now we have the environment ready. In the next section, we'll get down to JavaScript. Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

```
Object.assign(this.page, { "url": "https://javascript.info/devtools", "identifier": "Vdevtools" }); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true;
```

TutorialThe JavaScript language{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement": [{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"}], [{"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"}]}}JavaScript FundamentalsLet's learn the fundamentals of script building.

Hello, world!Code structureThe modern mode, "use strict"VariablesData typesInteraction: alert, prompt, confirmType ConversionsBasic operators, mathsComparisonsConditional branching: if, '?'Logical operatorsNullish coalescing operator '??'Loops: while and forThe "switch" statementFunctionsFunction expressionsArrow functions, the basicsJavaScript specialsCtrl + !•Previous lessonCtrl + !•Next lessonShareTutorial map

TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement": [{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"}], [{"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"}], [{"@type":"ListItem", "position":3, "name":"JavaScript Fundamentals", "item":"https://javascript.info/first-steps"}]}}November 1, 2021Hello, 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). We'll focus on JavaScript in the browser in the next part of the tutorial.

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” tagJavaScript 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 markupThe `<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>.

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

TasksShow an alertimportance: 5Create 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.

Demo in new windowsolution

```
<!DOCTYPE html>
<html>

<body>

  <script>
    alert( "I'm JavaScript!" );
  </script>

</body>

</html>
```

Open the solution in a sandbox.Show an alert with an external scriptimportance: 5Take the solution of the previous task Show an 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.

solutionThe HTML code:

```

    <!DOCTYPE html>
<html>

<body>

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

</body>

</html>

```

For the file alert.js in the same folder:

```

alert("I'm JavaScript!");

```

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/hello-world", "identifier": "Vhello-world" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}]};May 29, 2021Code structureThe first thing we'll study is the building blocks of code. StatementsStatements 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');
```

SemicolonsA 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.

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.

CommentsAs 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.

Later in the tutorial there will be a chapter Code quality that also explains how to write better comments.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/structure","identifier":"Vstructure"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}],{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://

javascript.info/js"}},{ "@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}}]May 19, 2020The 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 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.

All examples in this tutorial assume strict mode unless (very rarely) specified otherwise.
Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://www.javascript.info/strict-mode", "identifier": "strict-mode" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://www.javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://www.javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://www.javascript.info/first-steps"}]}November 16, 2022VariablesMost of the time, a JavaScript application needs to work with information. Here are two examples:

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

Variables are used to store this information.

A variableA variable 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”](#).

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 so-called pure functional programming languages, such as Haskell, 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.

Variable naming There are two limitations on variable names in JavaScript:

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

Examples of valid names:

```
let userName;  
let test123;
```

When the name contains multiple words, 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, Chinese logograms and so on, like this:

```
let C„<Dò Ò râââs°  
let b = '...';
```

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, 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
```

ConstantsTo 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.

When 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"](#), 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

Declare two variables: `admin` and `name`.

Assign the value `"John"` to `name`.

Copy the value from `name` to `admin`.

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

Create a variable with the name of our planet. How would you name such a variable?

Create a variable to store the name of a current visitor to a website. How would you

name that variable?

solutionThe variable for our planetThe variable for our planetThat'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 visitorThe name of the current visitor

```
let currentUser = "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.
And if your editor does not have proper autocomplete, get a new one.
Uppercase const?importance: 4Examine 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.

Ctrl + ! • Previous lesson Ctrl + ! • Next lesson Share Tutorial mapComments read this before commenting... If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/variables", "identifier": "variables" }); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true; Tutorial The JavaScript language JavaScript Fundamentals { "@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}] } November 1, 2022 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 ". 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.

`BigInt`In JavaScript, the “number” type cannot safely represent integer values larger than $(2^{53}-1)$ (that's 9007199254740991), or less than $-(2^{53}-1)$ for negatives.

To be really precise, the “number” type can store larger integers (up to $1.7976931348623157 \times 10^{308}$), but outside of the safe integer range $\pm(2^{53}-1)$ 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 $(2^{53}-1)$ can't be stored at all in the “number” type.

For most purposes $\pm(2^{53}-1)$ 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`. Read it when you need such big numbers.

Compatibility issues

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

You can check MDN BigInt compatibility table to know which versions of a browser are supported.

StringA 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.

Double quotes: "Hello".

Single quotes: 'Hello'.

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.

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")
```

We'll cover booleans more deeply in the chapter Logical operators.
The “null” valueThe 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” valueThe 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, 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 operand. 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:

`Math` is a built-in object that provides mathematical operations. We will learn it in the chapter `Numbers`. Here, it serves just as an example of an object.

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.

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 $\pm(2^{53}-1)$.

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}` ); // ?
```

solutionBackticks 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
```

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/types","identifier":"types"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"{"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]}September 8, 2020Interaction: alert, prompt, confirmAs we'll be using the browser as our demo environment, let's see a couple of functions to interact with the user: alert, prompt and confirm.alertThis one we've seen already. It shows a message and waits for the user to press "OK".For example:

```
alert("Hello");
```

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

```
result = prompt(title, [default]);
```

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

title

The text to show the visitor.

default

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

The square brackets in syntax [...]

The square brackets around default in the syntax above denote that the parameter is optional, not required.

The visitor can type something in the prompt input field and press OK. Then we get that text in the result. Or they can cancel the input by pressing Cancel or hitting the Esc key, then we get null as the result.

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

```
let age = prompt('How old are you?', 100);  
alert(`You are ${age} years old!`); // You are 100 years old!
```

In IE: always supply a default
The second parameter is optional, but if we don't supply it, Internet Explorer will insert the text "undefined" into the prompt.
Run this code in Internet Explorer to see:

```
let test = prompt("Test");
```

So, for prompts to look good in IE, we recommend always providing the second argument:

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

confirmThe syntax:

```
result = confirm(question);
```

The function confirm shows a modal window with a question and two buttons: OK and Cancel.

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

For example:

```
let isBoss = confirm("Are you the boss?");  
  
alert( isBoss ); // true if OK is pressed
```

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

alert

shows a message.

prompt

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

confirm

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

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

There are two limitations shared by all the methods above:

The exact location of the modal window is determined by the browser. Usually, it’s in the center.

The exact look of the window also depends on the browser. We can’t modify it.

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

TasksA simple pageimportance: 4Create a web-page that asks for a name and outputs it.

Run the demosolutionJavaScript-code:

```
let name = prompt("What is your name?", "");  
alert(name);
```

The full page:

```
<!DOCTYPE html>  
<html>
```

```

<body>

<script>
  'use strict';

  let name = prompt("What is your name?", "");
  alert(name);
</script>

</body>
</html>

```

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/alert-prompt-confirm", "identifier": "Alert-prompt-confirm" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement": [{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"}, {"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"}, {"@type":"ListItem", "position":3, "name":"JavaScript Fundamentals", "item":"https://javascript.info/first-steps"}]}

January 24, 2023Type ConversionsMost of the time, operators and functions automatically convert the values given to them to the right type.

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

There are also cases when we need to explicitly convert a value to the expected type.

Not talking about objects yet

In this chapter, we won't cover objects. For now, we'll just be talking about primitives.

Later, after we learn about objects, in the chapter Object to primitive conversion we'll see how objects fit in.

String ConversionString conversion happens when we need the string form of a value. For example, alert(value) does it to show the value.

We can also call the String(value) function to convert a value to a string:

```
let value = true;  
alert(typeof value); // boolean
```

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

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

Numeric Conversion Numeric conversion in mathematical functions and expressions happens automatically.

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

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

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

```
    let str = "123";  
    alert(typeof str); // string  
  
    let num = Number(str); // becomes a number 123  
  
    alert(typeof num); // number
```

Explicit conversion is usually required when we read a value from a string-based source like a text form but expect a number to be entered.
If the string is not a valid number, the result of such a conversion is NaN. For instance:

```
    let age = Number("an arbitrary string instead of a number");  
  
    alert(age); // NaN, conversion failed
```

Numeric conversion rules:

Value	Becomes...
-------	------------

undefined
NaN

null
0

true and false
1 and 0

string
Whitespaces (includes spaces, tabs \t, newlines \n etc.) from the start and end are removed. If the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN.

Examples:

```
    alert( Number(" 123 ") ); // 123
alert( Number("123z") );    // NaN (error reading a number at "z")
alert( Number(true) );      // 1
alert( Number(false) );     // 0
```

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

Most mathematical operators also perform such conversion, we'll see that in the next chapter.

Boolean Conversion Boolean conversion is the simplest one.

It happens in logical operations (later we'll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

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

Other values become true.

For instance:

```
    alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
```

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

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

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

```
    alert( Boolean("0") ); // true
alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)
```

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

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

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

The conversion follows the rules:

Value	Becomes...
-------	------------

undefined	NaN
-----------	-----

null	0
------	---

true / false	1 / 0
--------------	-------

string	The string is read “as is”, whitespaces (includes spaces, tabs \t, newlines \n etc.) from both sides are ignored. An empty string becomes 0. An error gives NaN.
--------	--

Boolean Conversion – Occurs in logical operations. Can be performed with `Boolean(value)`. Follows the rules:

Value	Becomes...
-------	------------

0, null, undefined, NaN, ""
false

any other value
true

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

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

Objects aren't covered here. We'll return to them later in the chapter Object to primitive conversion that is devoted exclusively to objects after we learn more basic things about JavaScript.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://javascript.info/type-conversions","identifier":"type-conversions"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;

TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]]November 14, 2022Basic operators, mathsWe know many operators from school. They are things like addition +, multiplication *, subtraction -, and so on.

In this chapter, we'll start with simple operators, then concentrate on JavaScript-specific aspects, not covered by school arithmetic.

Terms: “unary”, “binary”, “operand”Before we move on, let's grasp some common terminology.

An operand – is what operators are applied to. For instance, in the multiplication of $5 * 2$ there are two operands: the left operand is 5 and the right operand is 2. Sometimes,

people call these “arguments” instead of “operands”.

An operator is unary if it has a single operand. For example, the unary negation - reverses the sign of a number:

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

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

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

Formally, in the examples above we have two different operators that share the

same symbol: the negation operator, a unary operator that reverses the sign, and the subtraction operator, a binary operator that subtracts one number from another.

MathsThe following math operations are supported:

Addition +,
Subtraction -,
Multiplication *,
Division /,
Remainder %,
Exponentiation **.

The first four are straightforward, while % and ** need a few words about them.

Remainder %The remainder operator %, despite its appearance, is not related to percents.

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

For instance:

```
alert( 5 % 2 ); // 1, the remainder of 5 divided by 2
alert( 8 % 3 ); // 2, the remainder of 8 divided by 3
alert( 8 % 4 ); // 0, the remainder of 8 divided by 4
```

Exponentiation **The exponentiation operator $a ** b$ raises a to the power of b . In school maths, we write that as ab .
For instance:

```
    alert( 2 ** 2 ); // 22 = 4  
alert( 2 ** 3 ); // 23 = 8  
alert( 2 ** 4 ); // 24 = 16
```

Just like in maths, the exponentiation operator is defined for non-integer numbers as well.

For example, a square root is an exponentiation by $\frac{1}{2}$:

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

String concatenation with binary + Let's meet the features of JavaScript operators that are beyond school arithmetics.

Usually, the plus operator + sums numbers.

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

```
    let s = "my" + "string";  
    alert(s); // mystring
```

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

For example:

```
    alert( '1' + 2 ); // "12"  
alert( 2 + '1' ); // "21"
```

See, it doesn't matter whether the first operand is a string or the second one. Here's a more complex example:

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

Here, operators work one after another. The first + sums two numbers, so it returns 4, then the next + adds the string 1 to it, so it's like $4 + '1' = '41'$.

```
alert('1' + 2 + 2); // "122" and not "14"
```

Here, the first operand is a string, the compiler treats the other two operands as strings too. The 2 gets concatenated to '1', so it's like '1' + 2 = "12" and "12" + 2 = "122". The binary + is the only operator that supports strings in such a way. Other arithmetic operators work only with numbers and always convert their operands to numbers. Here's the demo for subtraction and division:

```
alert( 6 - '2' ); // 4, converts '2' to a number  
alert( '6' / '2' ); // 3, converts both operands to numbers
```

Numeric conversion, unary +The plus + exists in two forms: the binary form that we used above and the unary form.

The unary plus or, in other words, the plus operator + applied to a single value, doesn't do anything to numbers. But if the operand is not a number, the unary plus converts it into a number.

For example:

```
// No effect on numbers  
let x = 1;  
alert( +x ); // 1
```

```
let y = -2;  
alert( +y ); // -2
```

```
// Converts non-numbers  
alert( +true ); // 1  
alert( +"" ); // 0
```

It actually does the same thing as `Number(...)`, but is shorter.
The need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, they are usually strings. What if we want to sum them?
The binary plus would add them as strings:

```
let apples = "2";  
let oranges = "3";  
  
alert( apples + oranges ); // "23", the binary plus concatenates strings
```

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

```

    let apples = "2";
    let oranges = "3";

    // both values converted to numbers before the binary plus
    alert( +apples + +oranges ); // 5

    // the longer variant
    // alert( Number(apples) + Number(oranges) ); // 5

```

From a mathematician's standpoint, the abundance of pluses may seem strange. But from a programmer's standpoint, there's nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up. Why are unary pluses applied to values before the binary ones? As we're going to see, that's because of their higher precedence.

Operator precedence If an expression has more than one operator, the execution order is defined by their precedence, or, in other words, the default priority order of operators. From school, we all know that the multiplication in the expression $1 + 2 * 2$ should be calculated before the addition. That's exactly the precedence thing. The multiplication is said to have a higher precedence than the addition.

Parentheses override any precedence, so if we're not satisfied with the default order, we can use them to change it. For example, write $(1 + 2) * 2$.

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

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

Precedence
Name
Sign

...
...
...

14
unary plus
+

14
unary negation
-

13
exponentiation
**

12
multiplication
*

12
division
/

11
addition
+

11
subtraction
-

...
...
...

2
assignment
=

...
...
...

As we can see, the “unary plus” has a priority of 14 which is higher than the 11 of “addition” (binary plus). That’s why, in the expression “+apples + +oranges”, unary pluses work before the addition.

AssignmentLet’s note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 2.

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

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

```
alert( x ); // 5
```

Assignment = returns a valueThe fact of = being an operator, not a “magical” language construct has an interesting implication.

All operators in JavaScript return a value. That’s obvious for + and -, but also true for =. The call `x = value` writes the value into x and then returns it.

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

```
let a = 1;
```

```
let b = 2;
```

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

```
alert( a ); // 3  
alert( c ); // 0
```

In the example above, the result of expression $(a = b + 1)$ is the value which was assigned to a (that is 3). It is then used for further evaluations. Funny code, isn't it? We should understand how it works, because sometimes we see it in JavaScript libraries.

Although, please don't write the code like that. Such tricks definitely don't make code clearer or readable.

Chaining assignments Another interesting feature is the ability to chain assignments:

```
let a, b, c;  
  
a = b = c = 2 + 2;  
  
alert( a ); // 4  
alert( b ); // 4  
alert( c ); // 4
```

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

Once again, for the purposes of readability it's better to split such code into few lines:

```
c = 2 + 2;  
b = c;  
a = c;
```

That's easier to read, especially when eye-scanning the code fast.
Modify-in-place We often need to apply an operator to a variable and store the new result in that same variable.
For example:

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

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

```
let n = 2;  
n += 5; // now n = 7 (same as n = n + 5)  
n *= 2; // now n = 14 (same as n = n * 2)  
  
alert( n ); // 14
```

Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: `/`, `=`, `-=`, etc.
Such operators have the same precedence as a normal assignment, so they run after most other calculations:

```
let n = 2;  
  
n *= 3 + 5; // right part evaluated first, same as n *= 8  
  
alert( n ); // 16
```

Increment/decrement
Increasing or decreasing a number by one is among the most common numerical operations.
So, there are special operators for it:

Increment ++ increases a variable by 1:

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

Decrement -- decreases a variable by 1:

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

Important:

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

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

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

Both of these statements do the same thing: increase counter by 1.

Is there any difference? Yes, but we can only see it if we use the returned value of ++/--. Let's clarify. As we know, all operators return a value. Increment/decrement is no exception. The prefix form returns the new value while the postfix form returns the old value (prior to increment/decrement).

To see the difference, here's an example:

```
let counter = 1;
```

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

```
alert(a); // 2
```

In the line (*), the prefix form `++counter` increments `counter` and returns the new value, 2. So, the alert shows 2.
Now, let's use the postfix form:

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

```
alert(a); // 1
```

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

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

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

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

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

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

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

Increment/decrement among other operators

The operators ++/-- can be used inside expressions as well. Their precedence is higher than most other arithmetical operations.

For instance:

```
let counter = 1;  
alert( 2 * ++counter ); // 4
```

Compare with:

```
let counter = 1;  
alert( 2 * counter++ ); // 2, because counter++ returns the "old" value
```

Though technically okay, such notation usually makes code less readable. One line

does multiple things – not good.

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

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

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

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

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

The list of operators:

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

These operators are used very rarely, when we need to fiddle with numbers on the very lowest (bitwise) level. We won’t need these operators any time soon, as web development has little use of them, but in some special areas, such as cryptography, they are useful. You can read the Bitwise Operators chapter on MDN when a need arises.

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

The comma operator allows us to evaluate several expressions, dividing them with a

comma `,`. Each of them is evaluated but only the result of the last one is returned. For example:

```
let a = (1 + 2, 3 + 4);  
alert( a ); // 7 (the result of 3 + 4)
```

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

Comma has a very low precedence

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

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

Why do we need an operator that throws away everything except the last expression? Sometimes, people use it in more complex constructs to put several actions in one line. For example:

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

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

using them.

TasksThe postfix and prefix formsimportance: 5What are the final values of all variables a, b, c and d after the code below?

```
let a = 1, b = 1;
```

```
let c = ++a; // ?
```

```
let d = b++; // ?
```

solutionThe answer is:

```
a = 2
```

```
b = 2
```

```
c = 2
```

```
d = 1
```

```
let a = 1, b = 1;
```

```
alert( ++a ); // 2, prefix form returns the new value
```

```
alert( b++ ); // 1, postfix form returns the old value
```

```
alert( a ); // 2, incremented once
```

```
alert( b ); // 2, incremented once
```

Assignment resultimportance: 3What are the values of a and x after the code below?

let a = 2;

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

solutionThe answer is:

a = 4 (multiplied by 2)

x = 5 (calculated as 1 + 4)

Type conversionsimportance: 5What are results of these expressions?

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

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

"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6

```
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
" -9 " + 5 = " -9 5" // (3)
" -9 " - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
" \t \n" - 2 = -2 // (7)
```

The addition with a string "" + 1 converts 1 to a string: "" + 1 = "1", and then we have "1" + 0, the same rule is applied.

The subtraction - (like most math operations) only works with numbers, it converts an empty string "" to 0.

The addition with a string appends the number 5 to the string.

The subtraction always converts to numbers, so it makes " -9 " a number -9 (ignoring spaces around it).

null becomes 0 after the numeric conversion.

undefined becomes NaN after the numeric conversion.

Space characters are trimmed off string start and end when a string is converted to a number. Here the whole string consists of space characters, such as \t, \n and a “regular” space between them. So, similarly to an empty string, it becomes 0.

Fix the additionimportance: 5Here's a code that asks the user for two numbers and shows their sum.

It works incorrectly. The output in the example below is 12 (for default prompt values). Why? Fix it. The result should be 3.

```
let a = prompt("First number?", 1);
let b = prompt("Second number?", 2);

alert(a + b); // 12
```

solutionThe reason is that prompt returns user input as a string.
So variables have values "1" and "2" respectively.

```
let a = "1"; // prompt("First number?", 1);  
let b = "2"; // prompt("Second number?", 2);  
  
alert(a + b); // 12
```

What we should do is to convert strings to numbers before +. For example, using
Number() or prepending them with +.
For example, right before prompt:

```
let a = +prompt("First number?", 1);  
let b = +prompt("Second number?", 2);  
  
alert(a + b); // 3
```

Or in the alert:

```
let a = prompt("First number?", 1);
let b = prompt("Second number?", 2);

alert(+a + +b); // 3
```

Using both unary and binary + in the latest code. Looks funny, doesn't it?

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/operators","identifier":"V operators"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true; TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}], [{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https:// javascript.info/js"}], [{"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]}October 1, 2021ComparisonsWe 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: $a \geq b$, $a \leq b$.

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 \neq , but in JavaScript it's written as $a !== b$.

In this article we'll learn more about different types of comparisons, how JavaScript makes them, including important peculiarities.

At the end you'll find a good recipe to avoid "JavaScript quirks"-related issues.
Boolean is the resultAll 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 comparisonTo 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:

Compare the first character of both strings.

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.

Otherwise, if both strings' first characters are the same, compare the second characters the same way.

Repeat until the end of either string.

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:

G is the same as G.

l is the same as l.

o is greater than e. Stop here. The first string is greater.

Not a real dictionary, but Unicode order

The comparison algorithm given above is roughly equivalent to the one used in 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.

Comparison of different typesWhen 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 equalityA 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 problemsWhy 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.

TasksComparisonsimportance: 5What 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:

Obviously, true.

Dictionary comparison, hence false. "a" is smaller than "p".

Again, dictionary comparison, first char "2" is greater than the first char "1".

Values null and undefined equal each other only.

Strict equality is strict. Different types from both sides lead to false.

Similar to (4), null only equals undefined.

Strict equality of different types.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/comparison","identifier":"Vcomparison"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true; TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]}December 7, 2022Conditional 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" statementThe 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**:

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” clauseThe 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” 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 ‘?’ 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 '?' 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:

The first question mark checks whether `age < 3`.
 If true – it returns 'Hi, baby!'. Otherwise, it continues to the expression after the colon “:”, checking `age < 18`.
 If that's true – it returns 'Hello!'. Otherwise, it continues to the expression after the next colon “:”, checking `age < 100`.
 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 ‘?’ 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 it when you need to execute different branches of code.

Task: 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 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!"

Demo in new window 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 importance: 2 Using if..else, write the code which gets a number via prompt and then shows in alert:

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

In this task we assume that the input is always a number.
Demo in new window 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 '?'importance: 5Rewrite 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 '?'importance: 5Rewrite 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' :
'';
```

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/vifelse", "identifier": "vifelse" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}]}June 5, 2022Logical operatorsThere 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) 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 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”.

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.

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) 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
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, it 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)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:

Converts the operand to boolean type: true/false.
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 ||.

TasksWhat's the result of OR?importance: 5What is the code below going to output?

```
alert( null || 2 || undefined );
```

solutionThe answer is 2, that's the first truthy value.

```
alert( null || 2 || undefined );
```

What's the result of OR'ed alerts?importance: 3What will the code below output?

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

solutionThe 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.

The first OR || evaluates its left operand alert(1). That shows the first message with 1.

The alert returns undefined, so OR goes on to the second operand searching for a truthy value.

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?importance: 5What is this code going to show?

```
alert( 1 && null && 2 );
```

solutionThe answer: null, because it's the first falsy value from the list.

```
alert(1 && null && 2);
```

What is the result of AND'ed alerts?importance: 3What will this code show?

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

solutionThe 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 ORimportance: 5What will the result be?

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

solutionThe 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 betweenimportance: 3Write 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 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" 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 importance: 3Write the code which asks for a login with prompt.
 If the visitor enters "Admin", then prompt for a password, if the input is an empty line or
 Esc – show “Canceled”, if it’s another string – then show “I don’t know you”.
 The password is checked as follows:

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

The schema:

Please use nested if blocks. Mind the overall readability of the code.
 Hint: passing an empty input to a prompt returns an empty string ". Pressing ESC
 during a prompt returns null.
 Run the demosolution

```

    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.

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/logical-operators","identifier":"logical-operators"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"{"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]}April 23, 2023Nullish

coalescing operator '??'

A recent addition

This is a recent addition to the language.

Old browsers may need polyfills.

The nullish coalescing operator is written as two question marks ??.

As it treats null and undefined similarly, we'll use a special term here, in this article. For brevity, we'll say that a value is "defined" when it's neither null nor undefined.

The result of a ?? b is:

if a is defined, then a,

if a isn't defined, then b.

In other words, ?? returns the first argument if it's not null/undefined. Otherwise, the second one.

The nullish coalescing operator isn't anything completely new. It's just a nice syntax to get the first "defined" value of the two.

We can rewrite `result = a ?? b` using the operators that we already know, like this:

```
result = (a !== null && a !== undefined) ? a : b;
```

Now it should be absolutely clear what ?? does. Let's see where it helps.

The common use case for ?? is to provide a default value.

For example, here we show user if its value isn't null/undefined, otherwise Anonymous:

```
let user;
```

```
alert(user ?? "Anonymous"); // Anonymous (user is undefined)
```

Here's the example with user assigned to a name:

```
let user = "John";  
  
alert(user ?? "Anonymous"); // John (user is not null/undefined)
```

We can also use a sequence of ?? to select the first value from a list that isn't null/undefined.

Let's say we have a user's data in variables firstName, lastName or nickName. All of them may be not defined, if the user decided not to fill in the corresponding values. We'd like to display the user name using one of these variables, or show "Anonymous" if all of them are null/undefined. Let's use the ?? operator for that:

```
let firstName = null;  
let lastName = null;  
let nickName = "Supercoder";  
  
// shows the first defined value:  
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

Comparison with || The OR || operator can be used in the same way as ??, as it was described in the previous chapter.
For example, in the code above we could replace ?? with || and still get the same result:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// shows the first truthy value:
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

Historically, the OR || operator was there first. It's been there since the beginning of JavaScript, so developers were using it for such purposes for a long time.
On the other hand, the nullish coalescing operator ?? was added to JavaScript only recently, and the reason for that was that people weren't quite happy with ||.
The important difference between them is that:

|| returns the first truthy value.
?? returns the first defined value.

In other words, || doesn't distinguish between false, 0, an empty string "" and null/undefined. They are all the same – falsy values. If any of these is the first argument of ||, then we'll get the second argument as the result.
In practice though, we may want to use default value only when the variable is null/undefined. That is, when the value is really unknown/not set.
For example, consider this:

```
let height = 0;  
  
alert(height || 100); // 100  
alert(height ?? 100); // 0
```

The `height || 100` checks `height` for being a falsy value, and it's 0, falsy indeed.
so the result of `||` is the second argument, 100.

The `height ?? 100` checks `height` for being null/undefined, and it's not,
so the result is `height` "as is", that is 0.

In practice, the zero height is often a valid value, that shouldn't be replaced with the default. So `??` does just the right thing.

Precedence The precedence of the `??` operator is the same as `||`. They both equal 3 in the MDN table.

That means that, just like `||`, the nullish coalescing operator `??` is evaluated before `=` and `?`, but after most other operations, such as `+`, `*`.

So we may need to add parentheses in expressions like this:


```
    let height = null;
let width = null;

// important: use parentheses
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Otherwise, if we omit parentheses, then as `*` has the higher precedence than `??`, it would execute first, leading to incorrect results.

```
// without parentheses
let area = height ?? 100 * width ?? 50;

// ...works this way (not what we want):
let area = height ?? (100 * width) ?? 50;
```

Using `??` with `&&` or `||` Due to safety reasons, JavaScript forbids using `??` together with `&&` and `||` operators, unless the precedence is explicitly specified with parentheses. The code below triggers a syntax error:

```
let x = 1 && 2 ?? 3; // Syntax error
```

The limitation is surely debatable, it was added to the language specification with the purpose to avoid programming mistakes, when people start to switch from `||` to `??`. Use explicit parentheses to work around it:

```
let x = (1 && 2) ?? 3; // Works  
alert(x); // 2
```

Summary

The nullish coalescing operator `??` provides a short way to choose the first “defined” value from a list.

It’s used to assign default values to variables:

```
// set height=100, if height is null or undefined  
height = height ?? 100;
```

The operator `??` has a very low precedence, only a bit higher than `?` and `=`, so consider adding parentheses when using it in an expression.

It’s forbidden to use it with `||` or `&&` without explicit parentheses.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

```
codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/voidish-coalescing-  
operator", "identifier": "voidish-coalescing-operator" }); };var DisqusShortname =  
"javascriptinfo";var DisqusEnabled = true;  
TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://  
schema.org", "@type": "BreadcrumbList", "itemListElement":  
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript  
Fundamentals", "item": "https://javascript.info/first-steps"}]}June 19, 2022Loops: while  
and forWe 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 to loop over object properties.

See for...of and iterables for looping over arrays and iterable objects.

Otherwise, please read on.

The “while” loopThe 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” loopThe 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” loopThe 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 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

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 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 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.

SummaryWe 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.

TasksLast loop valueimportance: 3What is the last value alerted by this code? Why?

```
let i = 3;

while (i) {
  alert( i-- );
}
```

solutionThe 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?importance: 4For every loop iteration, write down which value it outputs and then compare it with the solution.
Both loops alert the same values, or not?

The prefix form `++i`:

```
let i = 0;
while (++i < 5) alert( i );
```

The postfix form `i++`

```
let i = 0;
while (i++ < 5) alert( i );
```

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

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 $\text{while}(5 < 5)$ fails, and the loop stops. So 5 is not shown.

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 $\text{while}(4 < 5)$ – true, and the control goes on to alert. The value $i = 5$ is the last one, because on the next step $\text{while}(5 < 5)$ is false.

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?

The postfix form:

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

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 deduced from the algorithm of for:

Execute once `i = 0` before everything (begin).

Check the condition `i < 5`

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 importance: 5 Use the for loop to output even numbers from 2 to 10.

Run the demo 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"importance: 5Rewrite 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 correctimportance: 5Write 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.

Run the demoesolution

```
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:

The check for `num <= 100` – that is, the entered value is still not greater than 100. 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 numbersimportance: 3An integer number greater than 1 is called a prime 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.

solutionThere 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, General number field sieve etc.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

```
Object.assign(this.page, {"url":"https://javascript.info/while-for","identifier":"\\while-for"}); };var Disqus_shortcode = "javascriptinfo";var Disqus_enabled = true;
TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},{"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]]April 25, 2022The "switch" statementA switch statement can replace multiple if checks.
It gives a more descriptive way to compare a value with multiple variants.
The syntaxThe switch has one or more case blocks and an optional default.
It looks like this:
```

```
switch(x) {
case 'value1': // if (x === 'value1')
  ...
  [break]

case 'value2': // if (x === 'value2')
  ...
  [break]

default:
  ...
  [break]
}
```

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

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


```
    let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Too small' );
    break;
  case 4:
    alert( 'Exactly!' );
    break;
  case 5:
    alert( 'Too big' );
    break;
  default:
    alert( "I don't know such values" );
}
```

Here the switch starts to compare a from the first case variant that is 3. The match fails.
Then 4. That's a match, so the execution starts from case 4 until the nearest break.
If there is no break then the execution continues with the next case without any checks.
An example without break:

```
    let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Too small' );
```

```
case 4:
  alert( 'Exactly!' );
case 5:
  alert( 'Too big' );
default:
  alert( "I don't know such values" );
}
```

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

```
    alert( 'Exactly!' );
alert( 'Too big' );
alert( "I don't know such values" );
```

Any expression can be a switch/case argument
Both switch and case allow arbitrary expressions.
For example:

```
    let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("this runs, because +a is 1, exactly equals b+1");
    break;

  default:
```

```
    alert("this doesn't run");  
}
```

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

Grouping of “case” Several variants of case which share the same code can be grouped. For example, if we want the same code to run for case 3 and case 5:

```
    let a = 3;  
  
switch (a) {  
  case 4:  
    alert('Right!');  
    break;  
  
  case 3: // (*) grouped two cases  
  case 5:  
    alert('Wrong!');  
    alert("Why don't you take a math class?");  
    break;  
  
  default:  
    alert('The result is strange. Really.');
```

Now both 3 and 5 show the same message.
The ability to “group” cases is a side effect of how switch/case works without break. Here the execution of case 3 starts from the line (*) and goes through case 5, because there’s no break.
Type matters Let’s emphasize that the equality check is always strict. The values must

be of the same type to match.
For example, let's consider the code:

```
    let arg = prompt("Enter a value?");
switch (arg) {
  case '0':
  case '1':
    alert( 'One or zero' );
    break;

  case '2':
    alert( 'Two' );
    break;

  case 3:
    alert( 'Never executes!' );
    break;
  default:
    alert( 'An unknown value' );
}
```

For 0, 1, the first alert runs.

For 2 the second alert runs.

But for 3, the result of the prompt is a string "3", which is not strictly equal === to the number 3. So we've got a dead code in case 3! The default variant will execute.

Tasks Rewrite the "switch" into an "if" importance: 5 Write the code using if..else which would correspond to the following switch:

```

    switch (browser) {
case 'Edge':
    alert( "You've got the Edge!" );
    break;

case 'Chrome':
case 'Firefox':
case 'Safari':
case 'Opera':
    alert( 'Okay we support these browsers too' );
    break;

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

```

solutionTo precisely match the functionality of switch, the if must use a strict comparison '==='.
 For given strings though, a simple '==' works too.

```

    if(browser == 'Edge') {
    alert("You've got the Edge!");
} else if (browser == 'Chrome'
|| browser == 'Firefox'
|| browser == 'Safari'
|| browser == 'Opera') {
    alert( 'Okay we support these browsers too' );
} else {
    alert( 'We hope that this page looks ok!' );
}

```

Please note: the construct browser == 'Chrome' || browser == 'Firefox' ... is split into multiple lines for better readability.
 But the switch construct is still cleaner and more descriptive.
 Rewrite "if" into "switch"
 importance: 4
 Rewrite the code below using a single switch statement:

```

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

    if (a == 0) {
        alert( 0 );
    }
    if (a == 1) {
        alert( 1 );
    }

    if (a == 2 || a == 3) {
        alert( '2,3' );
    }

```

solutionThe first two checks turn into two case. The third check is split into two cases:

```

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

    switch (a) {
        case 0:
            alert( 0 );
            break;

        case 1:

```

```

    alert( 1 );
    break;

case 2:
case 3:
    alert( '2,3' );
    break;
}

```

Please note: the break at the bottom is not required. But we put it to make the code future-proof.

In the future, there is a chance that we'd want to add one more case, for example case 4. And if we forget to add a break before it, at the end of case 3, there will be an error. So that's a kind of self-insurance.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/switch", "identifier": "Vswitch" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}]}

October 14, 2022

FunctionsQuite 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 DeclarationTo 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 variablesA 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 variablesA 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 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 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 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 ??, 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 valueA 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 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 framework defines a function with \$. The Lodash library has its core function named _.

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

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 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.

SummaryA 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.
TasksIs "else" required?importance: 4The 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?

solutionNo difference!

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

Rewrite the function using '?' or '||'importance: 4The 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:

Using a question mark operator ?

Using OR ||

solutionUsing 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)` 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
```

solutionA 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)` importance: 4 Write a function `pow(x,n)` that returns x in power n . Or, in other words, multiplies x by itself n times and returns the result.

```
pow(3, 2) = 3 * 3 = 9  
pow(3, 3) = 3 * 3 * 3 = 27  
pow(1, 100) = 1 * 1 * ... * 1 = 1
```

Create a web-page that prompts for x and n , and then shows the result of `pow(x,n)`.
Run the demo 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) );
}

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://www.javascript.info/function-basics", "identifier": "function-basics" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript Fundamentals", "item": "https://javascript.info/first-steps"}]}July 14, 2022Function expressionsIn JavaScript, a function is not a “magical language structure”, but a special kind of value.

The syntax that we used before is called a Function Declaration:

```

function sayHi() {
    alert( "Hello" );
}

```

There is another syntax for creating a function that is called a Function Expression. It allows us to create a new function in the middle of any expression.

For example:

```
let sayHi = function() {  
  alert( "Hello" );  
};
```

Here we can see a variable sayHi getting a value, the new function, created as function() { alert("Hello"); }.

As the function creation happens in the context of the assignment expression (to the right side of =), this is a Function Expression.

Please note, there's no name after the function keyword. Omitting a name is allowed for Function Expressions.

Here we immediately assign it to the variable, so the meaning of these code samples is the same: "create a function and put it into the variable sayHi".

In more advanced situations, that we'll come across later, a function may be created and immediately called or scheduled for a later execution, not stored anywhere, thus remaining anonymous.

Function is a valueLet's reiterate: no matter how the function is created, a function is a value. Both examples above store a function in the sayHi variable.

We can even print out that value using alert:

```
function sayHi() {  
  alert( "Hello" );  
}
```

```
alert( sayHi ); // shows the function code
```

Please note that the last line does not run the function, because there are no

parentheses after sayHi. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

Surely, a function is a special value, in the sense that we can call it like sayHi().

But it's still a value. So we can work with it like with other kinds of values.

We can copy a function to another variable:

```
function sayHi() { // (1) create
  alert( "Hello" );
}

let func = sayHi; // (2) copy

func(); // Hello // (3) run the copy (it works)!
sayHi(); // Hello // this still works too (why wouldn't it)
```

Here's what happens above in detail:

The Function Declaration (1) creates the function and puts it into the variable named sayHi.

Line (2) copies it into the variable func. Please note again: there are no parentheses after sayHi. If there were, then func = sayHi() would write the result of the call sayHi() into func, not the function sayHi itself.

Now the function can be called as both sayHi() and func().

We could also have used a Function Expression to declare sayHi, in the first line:

```
let sayHi = function() { // (1) create
  alert( "Hello" );
```

```
};
```

```
let func = sayHi;  
// ...
```

Everything would work the same.

Why is there a semicolon at the end?

You might wonder, why do Function Expressions have a semicolon ; at the end, but Function Declarations do not:

```
function sayHi() {  
  // ...  
}
```

```
let sayHi = function() {  
  // ...  
};
```

The answer is simple: a Function Expression is created here as `function(...) {...}` inside the assignment statement: `let sayHi = ...;`. The semicolon ; is recommended at the end of the statement, it's not a part of the function syntax.

The semicolon would be there for a simpler assignment, such as `let sayHi = 5;`, and it's also there for a function assignment.

Callback functions Let's look at more examples of passing functions as values and using function expressions.

We'll write a function `ask(question, yes, no)` with three parameters:

```
question  
Text of the question  
yes  
Function to run if the answer is "Yes"  
no  
Function to run if the answer is "No"
```

The function should ask the question and, depending on the user's answer, call `yes()` or `no()`:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes()  
  else no();  
}
```

```
function showOk() {  
  alert( "You agreed." );  
}
```

```
function showCancel() {  
  alert( "You canceled the execution." );  
}
```

```
// usage: functions showOk, showCancel are passed as arguments to ask  
ask("Do you agree?", showOk, showCancel);
```

In practice, such functions are quite useful. The major difference between a real-life ask and the example above is that real-life functions use more complex ways to interact with the user than a simple confirm. In the browser, such functions usually draw a nice-looking question window. But that's another story.

The arguments showOk and showCancel of ask are called callback functions or just callbacks.

The idea is that we pass a function and expect it to be “called back” later if necessary. In our case, showOk becomes the callback for “yes” answer, and showCancel for “no” answer.

We can use Function Expressions to write an equivalent, shorter function:

```

        function ask(question, yes, no) {
        if (confirm(question)) yes()
        else no();
        }

    ask(
        "Do you agree?",
        function() { alert("You agreed."); },
        function() { alert("You canceled the execution."); }
    );

```

Here, functions are declared right inside the ask(...) call. They have no name, and so are called anonymous. Such functions are not accessible outside of ask (because they are not assigned to variables), but that's just what we want here. Such code appears in our scripts very naturally, it's in the spirit of JavaScript.

A function is a value representing an “action”

Regular values like strings or numbers represent the data.

A function can be perceived as an action.

We can pass it between variables and run when we want.

Function Expression vs Function Declaration Let's formulate the key differences between Function Declarations and Expressions.

First, the syntax: how to differentiate between them in the code.

Function Declaration: a function, declared as a separate statement, in the main code flow:

```

// Function Declaration
function sum(a, b) {
    return a + b;
}

```


Function Expression: a function, created inside an expression or inside another syntax construct. Here, the function is created on the right side of the “assignment expression”
=:

```
// Function Expression  
let sum = function(a, b) {  
  return a + b;  
};
```

The more subtle difference is when a function is created by the JavaScript engine. A Function Expression is created when the execution reaches it and is usable only from that moment.

Once the execution flow passes to the right side of the assignment `let sum = function...` – here we go, the function is created and can be used (assigned, called, etc.) from now on.

Function Declarations are different.

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

For example, this works:

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

The Function Declaration `sayHi` is created when JavaScript is preparing to start the script and is visible everywhere in it.
...If it were a Function Expression, then it wouldn't work:

```
sayHi("John"); // error!

let sayHi = function(name) { // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

Function Expressions are created when the execution reaches them. That would happen only in the line (*). Too late.
Another special feature of Function Declarations is their block scope.
In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.
For instance, let's imagine that we need to declare a function `welcome()` depending on the `age` variable that we get during runtime. And then we plan to use it some time later. If we use Function Declaration, it won't work as intended:

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

// conditionally declare a function
if (age < 18) {

    function welcome() {
        alert("Hello!");
    }

} else {

    function welcome() {
        alert("Greetings!");
    }

}

// ...use it later
welcome(); // Error: welcome is not defined
```

That's because a Function Declaration is only visible inside the code block in which it resides.
Here's another example:

```
    let age = 16; // take 16 as an example

if (age < 18) {
    welcome();          // \ (runs)
```

```

        // |
function welcome() { // |
    alert("Hello!"); // | Function Declaration is available
} // | everywhere in the block where it's declared
        // |
welcome(); // (runs)

} else {

    function welcome() {
        alert("Greetings!");
    }
}

// Here we're out of curly braces,
// so we can not see Function Declarations made inside of them.

welcome(); // Error: welcome is not defined

```

What can we do to make welcome visible outside of if?

The correct approach would be to use a Function Expression and assign welcome to the variable that is declared outside of if and has the proper visibility.

This code works as intended:

```

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

let welcome;

if (age < 18) {

    welcome = function() {
        alert("Hello!");
    };
}

```

```
} else {  
  
  welcome = function() {  
    alert("Greetings!");  
  };  
  
}  
  
welcome(); // ok now
```

Or we could simplify it even further using a question mark operator ?:

```
let age = prompt("What is your age?", 18);  
  
let welcome = (age < 18) ?  
  function() { alert("Hello!"); } :  
  function() { alert("Greetings!"); };  
  
welcome(); // ok now
```

When to choose Function Declaration versus Function Expression?

As a rule of thumb, when we need to declare a function, the first thing to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

That's also better for readability, as it's easier to look up function `f(...)` in the code than `let f = function(...) {...}`. Function Declarations are more "eye-catching".

...But if a Function Declaration does not suit us for some reason, or we need a conditional declaration (we've just seen an example), then Function Expression should be used.

Summary

Functions are values. They can be assigned, copied or declared in any place of the code.

If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".

If the function is created as a part of an expression, it's called a "Function Expression". Function Declarations are processed before the code block is executed. They are visible everywhere in the block.

Function Expressions are created when the execution flow reaches them.

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

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

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines –

wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};
```

```
Object.assign(this.page, { "url": "https://javascript.info/function-
```

```
expressions", "identifier": "function-expressions" }); };var DisqusShortname =
```

```
"javascriptinfo";var DisqusEnabled = true;
```

```
TutorialThe JavaScript languageJavaScript Fundamentals{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":
```

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},
```

```
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://
```

```
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "JavaScript
```

```
Fundamentals", "item": "https://javascript.info/first-steps"}]]}April 14, 2022Arrow functions,
```

the basicsThere's another very simple and concise syntax for creating functions, that's often better than Function Expressions.

It's called "arrow functions", because it looks like this:

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

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

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

```
    let func = function(arg1, arg2, ..., argN) {  
    return expression;  
};
```

Let's see a concrete example:

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

/* This arrow function is a shorter form of:

```
let sum = function(a, b) {  
    return a + b;  
};  
*/
```

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

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

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

```
    let double = n => n * 2;  
// roughly the same as: let double = function(n) { return n * 2 }  
  
alert( double(3) ); // 6
```

If there are no arguments, parentheses are empty, but they must be present:

```
    let sayHi = () => alert("Hello!");  
  
sayHi();
```

Arrow functions can be used in the same way as Function Expressions.
For instance, to dynamically create a function:


```

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

let welcome = (age < 18) ?
    () => alert('Hello!') :
    () => alert("Greetings!");

welcome();

```

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure. They are very convenient for simple one-line actions, when we're just too lazy to write many words.

Multiline arrow functions The arrow functions that we've seen so far were very simple. They took arguments from the left of `=>`, evaluated and returned the right-side expression with them.

Sometimes we need a more complex function, with multiple expressions and statements. In that case, we can enclose them in curly braces. The major difference is that curly braces require a return within them to return a value (just like a regular function does).

Like this:

```

    let sum = (a, b) => { // the curly brace opens a multiline function
let result = a + b;
return result; // if we use curly braces, then we need an explicit "return"
};

```

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

More to come

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

Arrow functions have other interesting features.

To study them in-depth, we first need to get to know some other aspects of JavaScript, so we'll return to arrow functions later in the chapter Arrow functions revisited.

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

Summary Arrow functions are handy for simple actions, especially for one-liners. They come in two flavors:

Without curly braces: (...args) => expression – the right side is an expression: the function evaluates it and returns the result. Parentheses can be omitted, if there's only a single argument, e.g. n => n*2.

With curly braces: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

Tasks Rewrite with arrow functions Replace Function Expressions with arrow functions in the code below:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes();  
  else no();  
}  
  
ask(  
  "Do you agree?",  
  function() { alert("You agreed."); },  
  function() { alert("You canceled the execution."); }  
);
```

solution

```
function ask(question, yes, no) {
  if (confirm(question)) yes();
  else no();
}

ask(
  "Do you agree?",
  () => alert("You agreed."),
  () => alert("You canceled the execution.")
);
```

Looks short and clean, right?

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://www.javascript.info/arrow-functions-basics","identifier":"arrow-functions-basics"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageJavaScript Fundamentals{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"JavaScript Fundamentals","item":"https://javascript.info/first-steps"}]}January 22, 2023JavaScript specialsThis chapter briefly recaps the features of JavaScript that we've learned by now, paying special attention to subtle moments.

Code structure Statements are delimited with a semicolon:

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

Usually, a line-break is also treated as a delimiter, so that would also work:

```
    alert('Hello')  
alert('World')
```

That's called "automatic semicolon insertion". Sometimes it doesn't work, for instance:

```
alert("There will be an error after this message")
```

```
[1, 2].forEach(alert)
```

Most codestyle guides agree that we should put a semicolon after each statement. Semicolons are not required after code blocks {...} and syntax constructs with them like loops:

```
function f() {  
  // no semicolon needed after function declaration  
}  
  
for(;;) {  
  // no semicolon needed after the loop  
}
```

...But even if we can put an “extra” semicolon somewhere, that’s not an error. It will be ignored.

More in: Code structure.

Strict modeTo fully enable all features of modern JavaScript, we should start scripts with "use strict".

```
'use strict';
```

```
...
```

The directive must be at the top of a script or at the beginning of a function body. Without "use strict", everything still works, but some features behave in the old-fashioned, “compatible” way. We’d generally prefer the modern behavior. Some modern features of the language (like classes that we’ll study in the future) enable strict mode implicitly.

More in: The modern mode, "use strict".
Variables Can be declared using:

let
const (constant, can't be changed)
var (old-style, will see later)

A variable name can include:

Letters and digits, but the first character may not be a digit.
Characters \$ and _ are normal, on par with letters.
Non-Latin alphabets and hieroglyphs are also allowed, but commonly not used.

Variables are dynamically typed. They can store any value:

```
let x = 5;  
x = "John";
```

There are 8 data types:

number for both floating-point and integer numbers,
bigint for integer numbers of arbitrary length,
string for strings,
boolean for logical values: true/false,
null – a type with a single value null, meaning “empty” or “does not exist”,
undefined – a type with a single value undefined, meaning “not assigned”,
object and symbol – for complex data structures and unique identifiers, we haven't learnt them yet.

The typeof operator returns the type for a value, with two exceptions:

```
typeof null == "object" // error in the language  
typeof function(){} == "function" // functions are treated specially
```

More in: Variables and Data types.
Interaction We're using a browser as a working environment, so basic UI functions will be:

`prompt(question, [default])`

Ask a question, and return either what the visitor entered or null if they clicked “cancel”.

`confirm(question)`

Ask a question and suggest to choose between Ok and Cancel. The choice is returned as true/false.

`alert(message)`

Output a message.

All these functions are modal, they pause the code execution and prevent the visitor from interacting with the page until they answer.

For instance:

```
let userName = prompt("Your name?", "Alice");  
let isTeaWanted = confirm("Do you want some tea?");
```

```
alert( "Visitor: " + userName ); // Alice  
alert( "Tea wanted: " + isTeaWanted ); // true
```

More in: Interaction: alert, prompt, confirm.

OperatorsJavaScript supports the following operators:

Arithmetical

Regular: `*` `+` `-` `/`, also `%` for the remainder and `**` for power of a number.

The binary plus `+` concatenates strings. And if any of the operands is a string, the other one is converted to string too:

```
    alert( '1' + 2 ); // '12', string  
alert( 1 + '2' ); // '12', string
```

Assignments

There is a simple assignment: `a = b` and combined ones like `a *= 2`.

Bitwise

Bitwise operators work with 32-bit integers at the lowest, bit-level: see the docs when they are needed.

Conditional

The only operator with three parameters: `cond ? resultA : resultB`. If `cond` is truthy, returns `resultA`, otherwise `resultB`.

Logical operators

Logical AND `&&` and OR `||` perform short-circuit evaluation and then return the value where it stopped (not necessary true/false). Logical NOT `!` converts the operand to boolean type and returns the inverse value.

Nullish coalescing operator

The `??` operator provides a way to choose a defined value from a list of variables. The result of `a ?? b` is `a` unless it's null/undefined, then `b`.

Comparisons

Equality check `==` for values of different types converts them to a number (except null and undefined that equal each other and nothing else), so these are equal:


```
    alert( 0 == false ); // true
alert( 0 == " "); // true
```

Other comparisons convert to a number as well.

The strict equality operator `===` doesn't do the conversion: different types always mean different values for it.

Values null and undefined are special: they equal `==` each other and don't equal anything else.

Greater/less comparisons compare strings character-by-character, other types are converted to a number.

Other operators

There are few others, like a comma operator.

More in: Basic operators, maths, Comparisons, Logical operators, Nullish coalescing operator '??'.

Loops

We covered 3 types of loops:

```
    // 1
while (condition) {
    ...
}
```

```
    // 2
do {
    ...
} while (condition);
```

```
    // 3
for(let i = 0; i < 10; i++) {
```

```
...  
}
```

The variable declared in `for(let...)` loop is visible only inside the loop. But we can also omit `let` and reuse an existing variable.

Directives `break/continue` allow to exit the whole loop/current iteration. Use labels to break nested loops.

Details in: Loops: `while` and `for`.

Later we'll study more types of loops to deal with objects.

The “switch” constructThe “switch” construct can replace multiple `if` checks. It uses `===` (strict equality) for comparisons.

For instance:

```
let age = prompt('Your age?', 18);  
  
switch (age) {  
  case 18:  
    alert("Won't work"); // the result of prompt is a string, not a number  
    break;  
  
  case "18":  
    alert("This works!");  
    break;  
  
  default:  
    alert("Any value not equal to one above");  
}
```

Details in: The "switch" statement.

Functions We covered three ways to create a function in JavaScript:

Function Declaration: the function in the main code flow

```
function sum(a, b) {  
  let result = a + b;  
  
  return result;  
}
```

Function Expression: the function in the context of an expression

```
let sum = function(a, b) {  
  let result = a + b;  
  
  return result;  
};
```

Arrow functions:

```
// expression on the right side  
let sum = (a, b) => a + b;
```

```
// or multi-line syntax with { ... }, need return here:
```

```

let sum = (a, b) => {
  // ...
  return a + b;
}

// without arguments
let sayHi = () => alert("Hello");

// with a single argument
let double = n => n * 2;

```

Functions may have local variables: those declared inside its body or its parameter list. Such variables are only visible inside the function.
Parameters can have default values: `function sum(a = 1, b = 2) {...}`.
Functions always return something. If there's no return statement, then the result is undefined.

Details: see Functions, Arrow functions, the basics.

More to come That was a brief list of JavaScript features. As of now we've studied only basics. Further in the tutorial you'll find more specials and advanced features of JavaScript.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/javascript-specials", "identifier": "V/javascript-specials" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript language{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}]}Code qualityThis chapter explains coding practices that we'll use further in the development.

Debugging in the browserCoding StyleCommentsNinja codeAutomated testing with MochaPolyfills and transpilersCtrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapTutorialThe JavaScript languageCode quality{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},

```
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"},{"@type":"ListItem","position":3,"name":"Code quality","item":"https://  
javascript.info/code-quality"}}}
```

June 26, 2022

Debugging in the browser

Before writing more complex code, let's talk about debugging.

Debugging is the process of finding and fixing errors within a script. All modern browsers and most other environments support debugging tools – a special UI in developer tools that makes debugging much easier. It also allows to trace the code step by step to see what exactly is going on.

We'll be using Chrome here, because it has enough features, most other browsers have a similar process.

The "Sources" panel

Your Chrome version may look a little bit different, but it still should be obvious what's there.

Open the example page in Chrome.

Turn on developer tools with F12 (Mac: Cmd+Opt+I).

Select the Sources panel.

Here's what you should see if you are doing it for the first time:

The toggler button opens the tab with files.

Let's click it and select hello.js in the tree view. Here's what should show up:

The Sources panel has 3 parts:

The File Navigator pane lists HTML, JavaScript, CSS and other files, including images that are attached to the page. Chrome extensions may appear here too.

The Code Editor pane shows the source code.

The JavaScript Debugging pane is for debugging, we'll explore it soon.

Now you could click the same toggler again to hide the resources list and give the code some space.

Console

If we press Esc, then a console opens below. We can type commands there and press Enter to execute.

After a statement is executed, its result is shown below.

For example, here `1+2` results in 3, while the function call `hello("debugger")` returns nothing, so the result is undefined:

BreakpointsLet's examine what's going on within the code of the example page. In hello.js, click at line number 4. Yes, right on the 4 digit, not on the code. Congratulations! You've set a breakpoint. Please also click on the number for line 8. It should look like this (blue is where you should click):

A breakpoint is a point of code where the debugger will automatically pause the JavaScript execution. While the code is paused, we can examine current variables, execute commands in the console etc. In other words, we can debug it. We can always find a list of breakpoints in the right panel. That's useful when we have many breakpoints in various files. It allows us to:

Quickly jump to the breakpoint in the code (by clicking on it in the right panel).
Temporarily disable the breakpoint by unchecking it.
Remove the breakpoint by right-clicking and selecting Remove.
...And so on.

Conditional breakpoints

Right click on the line number allows to create a conditional breakpoint. It only triggers when the given expression, that you should provide when you create it, is truthy. That's handy when we need to stop only for a certain variable value or for certain function parameters.

The command "debugger" We can also pause the code by using the debugger command in it, like this:

```
function hello(name) {  
  let phrase = `Hello, ${name}!`;   
  
  debugger; // <-- the debugger stops here  
  
  say(phrase);  
}
```

Such command works only when the development tools are open, otherwise the browser ignores it.

Pause and look aroundIn our example, `hello()` is called during the page load, so the easiest way to activate the debugger (after we've set the breakpoints) is to reload the page. So let's press F5 (Windows, Linux) or Cmd+R (Mac).

As the breakpoint is set, the execution pauses at the 4th line:

Please open the informational dropdowns to the right (labeled with arrows). They allow you to examine the current code state:

Watch – shows current values for any expressions.

You can click the plus + and input an expression. The debugger will show its value, automatically recalculating it in the process of execution.

Call Stack – shows the nested calls chain.

At the current moment the debugger is inside `hello()` call, called by a script in `index.html` (no function there, so it's called "anonymous").

If you click on a stack item (e.g. "anonymous"), the debugger jumps to the corresponding code, and all its variables can be examined as well.

Scope – current variables.

Local shows local function variables. You can also see their values highlighted right over the source.

Global has global variables (out of any functions).

There's also this keyword there that we didn't study yet, but we'll do that soon.

Tracing the executionNow it's time to trace the script.

There are buttons for it at the top of the right panel. Let's engage them.

– "Resume": continue the execution, hotkey F8.

Resumes the execution. If there are no additional breakpoints, then the execution just continues and the debugger loses control.

Here's what we can see after a click on it:

The execution has resumed, reached another breakpoint inside `say()` and paused there. Take a look at the “Call Stack” at the right. It has increased by one more call. We’re inside `say()` now.

- “Step”: run the next command, hotkey F9.

Run the next statement. If we click it now, alert will be shown.

Clicking this again and again will step through all script statements one by one.

- “Step over”: run the next command, but don’t go into a function, hotkey F10.

Similar to the previous “Step” command, but behaves differently if the next statement is a function call (not a built-in, like `alert`, but a function of our own).

If we compare them, the “Step” command goes into a nested function call and pauses the execution at its first line, while “Step over” executes the nested function call invisibly to us, skipping the function internals.

The execution is then paused immediately after that function call.

That’s good if we’re not interested to see what happens inside the function call.

- “Step into”, hotkey F11.

That’s similar to “Step”, but behaves differently in case of asynchronous function calls. If you’re only starting to learn JavaScript, then you can ignore the difference, as we don’t have asynchronous calls yet.

For the future, just note that “Step” command ignores async actions, such as `setTimeout` (scheduled function call), that execute later. The “Step into” goes into their code, waiting for them if necessary. See DevTools manual for more details.

- “Step out”: continue the execution till the end of the current function, hotkey Shift+F11.

Continue the execution and stop it at the very last line of the current function. That’s handy when we accidentally entered a nested call using `,` but it does not interest us, and we want to continue to its end as soon as possible.

- enable/disable all breakpoints.

That button does not move the execution. Just a mass on/off for breakpoints.

- enable/disable automatic pause in case of an error.

When enabled, if the developer tools is open, an error during the script execution automatically pauses it. Then we can analyze variables in the debugger to see what

went wrong. So if our script dies with an error, we can open debugger, enable this option and reload the page to see where it dies and what's the context at that moment.

Continue to here

Right click on a line of code opens the context menu with a great option called "Continue to here".

That's handy when we want to move multiple steps forward to the line, but we're too lazy to set a breakpoint.

LoggingTo output something to console from our code, there's console.log function. For instance, this outputs values from 0 to 4 to console:

```
// open console to see
for (let i = 0; i < 5; i++) {
  console.log("value,", i);
}
```

Regular users don't see that output, it is in the console. To see it, either open the Console panel of developer tools or press Esc while in another panel: that opens the console at the bottom.

If we have enough logging in our code, then we can see what's going on from the records, without the debugger.

SummaryAs we can see, there are three main ways to pause a script:

A breakpoint.

The debugger statements.

An error (if dev tools are open and the button is "on").

When paused, we can debug: examine variables and trace the code to see where the execution goes wrong.

There are many more options in developer tools than covered here. The full manual is

at <https://developers.google.com/web/tools/chrome-devtools>.

The information from this chapter is enough to begin debugging, but later, especially if you do a lot of browser stuff, please go there and look through more advanced capabilities of developer tools.

Oh, and also you can click at various places of dev tools and just see what's showing up. That's probably the fastest route to learn dev tools. Don't forget about the right click and context menus!

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/debugging-chrome", "identifier": "debugging-chrome" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageCode quality{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Code quality", "item": "https://javascript.info/code-quality"}]}June 26, 2022Coding StyleOur code must be as clean and easy to read as possible.

That is actually the art of programming – to take a complex task and code it in a way that is both correct and human-readable. A good code style greatly assists in that.

SyntaxHere is a cheat sheet with some suggested rules (see below for more details):

Now let's discuss the rules and reasons for them in detail.

There are no “you must” rules

Nothing is set in stone here. These are style preferences, not religious dogmas.

Curly BracesIn most JavaScript projects curly braces are written in “Egyptian” style with the opening brace on the same line as the corresponding keyword – not on a new line. There should also be a space before the opening bracket, like this:

```
    if (condition) {  
    // do this
```

```
// ...and that
// ...and that
}
```

A single-line construct, such as `if (condition) doSomething()`, is an important edge case. Should we use braces at all?

Here are the annotated variants so you can judge their readability for yourself:

Ø=⤵ Beginners sometimes do that. Bad! Curly braces are not needed:

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

Ø=⤵ Split to a separate line without braces. Never do that, easy to make an error when adding new lines:

```
if (n < 0)
  alert(`Power ${n} is not supported`);
```

Ø=⤵ One line without braces – acceptable, if it's short:

```
if (n < 0) alert(`Power ${n} is not supported`);
```

Ø=⤵ The best variant:

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
}
```

For a very brief code, one line is allowed, e.g. `if (cond) return null`. But a code block (the last variant) is usually more readable.

Line Length No one likes to read a long horizontal line of code. It's best practice to split them.

For example:

```
// backtick quotes ` allow to split the string into multiple lines
let str = `
  ECMA International's TC39 is a group of JavaScript developers,
  implementers, academics, and more, collaborating with the community
  to maintain and evolve the definition of JavaScript.
`;
```

And, for if statements:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

The maximum line length should be agreed upon at the team-level. It's usually 80 or 120 characters.

Indents There are two types of indents:

Horizontal indents: 2 or 4 spaces.

A horizontal indentation is made using either 2 or 4 spaces or the horizontal tab symbol (key Tab). Which one to choose is an old holy war. Spaces are more common nowadays.

One advantage of spaces over tabs is that spaces allow more flexible configurations of indents than the tab symbol.

For instance, we can align the parameters with the opening bracket, like this:

```
    show(parameters,
    aligned, // 5 spaces padding at the left
    one,
    after,
    another
  ) {
  // ...
}
```

Vertical indents: empty lines for splitting code into logical blocks.

Even a single function can often be divided into logical blocks. In the example below, the initialization of variables, the main loop and returning the result are split vertically:

```
    function pow(x, n) {
    let result = 1;
    //      <--
    for (let i = 0; i < n; i++) {
    result *= x;
    }
    //      <--
    return result;
    }
```

Insert an extra newline where it helps to make the code more readable. There should not be more than nine lines of code without a vertical indentation.

Semicolons A semicolon should be present after each statement, even if it could possibly be skipped.

There are languages where a semicolon is truly optional and it is rarely used. In

JavaScript, though, there are cases where a line break is not interpreted as a semicolon, leaving the code vulnerable to errors. See more about that in the chapter Code structure.

If you're an experienced JavaScript programmer, you may choose a no-semicolon code style like StandardJS. Otherwise, it's best to use semicolons to avoid possible pitfalls.

The majority of developers put semicolons.

Nesting Levels Try to avoid nesting code too many levels deep.

For example, in the loop, it's sometimes a good idea to use the continue directive to avoid extra nesting.

For example, instead of adding a nested if conditional like this:

```
    for (let i = 0; i < 10; i++) {  
  if (cond) {  
    ... // <- one more nesting level  
  }  
}
```

We can write:

```
    for (let i = 0; i < 10; i++) {  
  if (!cond) continue;  
  ... // <- no extra nesting level  
}
```

A similar thing can be done with if/else and return.

For example, two constructs below are identical.

Option 1:

```
function pow(x, n) {  
  if (n < 0) {  
    alert("Negative 'n' not supported");  
  } else {  
    let result = 1;  
    ...  
  }  
}
```

```

    for (let i = 0; i < n; i++) {
        result *= x;
    }

    return result;
}

```

Option 2:

```

    function pow(x, n) {
        if (n < 0) {
            alert("Negative 'n' not supported");
            return;
        }

        let result = 1;

        for (let i = 0; i < n; i++) {
            result *= x;
        }

        return result;
    }

```

The second one is more readable because the “special case” of $n < 0$ is handled early on. Once the check is done we can move on to the “main” code flow without the need for additional nesting.

Function Placement If you are writing several “helper” functions and the code that uses them, there are three ways to organize the functions.

Declare the functions above the code that uses them:

```
// function declarations
```

```
function createElement() {  
  ...  
}  
  
function setHandler(elem) {  
  ...  
}  
  
function walkAround() {  
  ...  
}  
  
// the code which uses them  
let elem = createElement();  
setHandler(elem);  
walkAround();
```

Code first, then functions

```
    // the code which uses the functions  
let elem = createElement();  
setHandler(elem);  
walkAround();  
  
// --- helper functions ---  
function createElement() {  
  ...  
}  
  
function setHandler(elem) {  
  ...  
}  
  
function walkAround() {  
  ...  
}
```


Mixed: a function is declared where it's first used.

Most of time, the second variant is preferred.

That's because when reading code, we first want to know what it does. If the code goes first, then it becomes clear from the start. Then, maybe we won't need to read the functions at all, especially if their names are descriptive of what they actually do.

Style GuidesA style guide contains general rules about "how to write" code, e.g. which quotes to use, how many spaces to indent, the maximal line length, etc. A lot of minor things.

When all members of a team use the same style guide, the code looks uniform, regardless of which team member wrote it.

Of course, a team can always write their own style guide, but usually there's no need to. There are many existing guides to choose from.

Some popular choices:

Google JavaScript Style Guide

Airbnb JavaScript Style Guide

Idiomatic.JS

StandardJS

(plus many more)

If you're a novice developer, start with the cheat sheet at the beginning of this chapter. Then you can browse other style guides to pick up more ideas and decide which one you like best.

Automated LintersLinters are tools that can automatically check the style of your code and make improving suggestions.

The great thing about them is that style-checking can also find some bugs, like typos in variable or function names. Because of this feature, using a linter is recommended even if you don't want to stick to one particular "code style".

Here are some well-known linting tools:

JSLint – one of the first linters.

JSHint – more settings than JSLint.

ESLint – probably the newest one.

All of them can do the job. The author uses ESLint.

Most linters are integrated with many popular editors: just enable the plugin in the editor and configure the style.

For instance, for ESLint you should do the following:

Install Node.js.

Install ESLint with the command `npm install -g eslint` (npm is a JavaScript package installer).

Create a config file named `.eslintrc` in the root of your JavaScript project (in the folder that contains all your files).
Install/enable the plugin for your editor that integrates with ESLint. The majority of editors have one.

Here's an example of an `.eslintrc` file:

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
    "indent": 2
  }
}
```

Here the directive "extends" denotes that the configuration is based on the "eslint:recommended" set of settings. After that, we specify our own. It is also possible to download style rule sets from the web and extend them instead. See <https://eslint.org/docs/user-guide/getting-started> for more details about installation. Also certain IDEs have built-in linting, which is convenient but not as customizable as ESLint.

Summary All syntax rules described in this chapter (and in the style guides referenced) aim to increase the readability of your code. All of them are debatable.

When we think about writing "better" code, the questions we should ask ourselves are: "What makes the code more readable and easier to understand?" and "What can help us avoid errors?" These are the main things to keep in mind when choosing and debating code styles.

Reading popular style guides will allow you to keep up to date with the latest ideas about code style trends and best practices.

Tasks Bad style importance: 4 What's wrong with the code style below?

```
function pow(x,n)
{
```

```

let result=1;
for(let i=0;i<n;i++) {result*=x;}
return result;
}

let x=prompt("x?",""), n=prompt("n?","")
if (n<=0)
{
  alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else
{
  alert(pow(x,n))
}

```

Fix it.
 solution You could note the following:

```

function pow(x,n) // <- no space between arguments
{ // <- figure bracket on a separate line
  let result=1; // <- no spaces before or after =
  for(let i=0;i<n;i++) {result*=x;} // <- no spaces
  // the contents of { ... } should be on a new line
  return result;
}

let x=prompt("x?",""), n=prompt("n?","") // <-- technically possible,
// but better make it 2 lines, also there's no spaces and missing ;
if (n<=0) // <- no spaces inside (n <= 0), and should be extra line above it
{ // <- figure bracket on a separate line
  // below - long lines can be split into multiple lines for improved readability
  alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else // <- could write it on a single line like "} else {"
{
  alert(pow(x,n)) // no spaces and missing ;
}

```

The fixed variant:

```

    function pow(x, n) {
    let result = 1;

    for (let i = 0; i < n; i++) {
        result *= x;
    }

    return result;
}

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

if (n <= 0) {
    alert(`Power ${n} is not supported,
    please enter an integer number greater than zero`);
} else {
    alert( pow(x, n) );
}

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {url:"https://javascript.info/coding-style","identifier":"coding-style"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true; TutorialThe JavaScript languageCode quality{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Code quality","item":"https://javascript.info/code-quality"}]}May 22, 2022CommentsAs we know from the chapter Code structure, comments can be single-line: starting with // and multiline: /* ... */. We normally use them to describe how and why the code works. At first sight, commenting might be obvious, but novices in programming often use them wrongly. Bad commentsNovices tend to use comments to explain “what is going on in the code”. Like this:

```
        // This code will do this thing (...) and that thing (...)
// ...and who knows what else...
very;
complex;
code;
```

But in good code, the amount of such “explanatory” comments should be minimal. Seriously, the code should be easy to understand without them. There’s a great rule about that: “if the code is so unclear that it requires a comment, then maybe it should be rewritten instead”. Recipe: factor out functions Sometimes it’s beneficial to replace a code piece with a function, like here:

```
function showPrimes(n) {
  nextPrime:
  for (let i = 2; i < n; i++) {

    // check if i is a prime number
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert(i);
  }
}
```

The better variant, with a factored out function isPrime:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
  }
}
```

```

    alert(i);
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }

  return true;
}

```

Now we can understand the code easily. The function itself becomes the comment. Such code is called self-descriptive.

Recipe: create functions And if we have a long “code sheet” like this:

```

    // here we add whiskey
for(let i = 0; i < 10; i++) {
  let drop = getWhiskey();
  smell(drop);
  add(drop, glass);
}

// here we add juice
for(let t = 0; t < 3; t++) {
  let tomato = getTomato();
  examine(tomato);
  let juice = press(tomato);
  add(juice, glass);
}

// ...

```

Then it might be a better variant to refactor it into functions like:

```

    addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}

```

Once again, functions themselves tell what's going on. There's nothing to comment. And also the code structure is better when split. It's clear what every function does, what it takes and what it returns.

In reality, we can't totally avoid "explanatory" comments. There are complex algorithms. And there are smart "tweaks" for purposes of optimization. But generally we should try to keep the code simple and self-descriptive.

Good commentsSo, explanatory comments are usually bad. Which comments are good?

Describe the architecture

Provide a high-level overview of components, how they interact, what's the control flow in various situations... In short – the bird's eye view of the code. There's a special language UML to build high-level architecture diagrams explaining the code. Definitely worth studying.

Document function parameters and usage

There's a special syntax JSDoc to document a function: usage, parameters, returned value.

For instance:

```

/**
 * Returns x raised to the n-th power.
 *
 * @param {number} x The number to raise.

```

```

* @param {number} n The power, must be a natural number.
* @return {number} x raised to the n-th power.
*/
function pow(x, n) {
  ...
}

```

Such comments allow us to understand the purpose of the function and use it the right way without looking in its code.

By the way, many editors like WebStorm can understand them as well and use them to provide autocomplete and some automatic code-checking.

Also, there are tools like JSDoc 3 that can generate HTML-documentation from the comments. You can read more information about JSDoc at <https://jsdoc.app>.

Why is the task solved this way?

What's written is important. But what's not written may be even more important to understand what's going on. Why is the task solved exactly this way? The code gives no answer.

If there are many ways to solve the task, why this one? Especially when it's not the most obvious one.

Without such comments the following situation is possible:

You (or your colleague) open the code written some time ago, and see that it's "suboptimal".

You think: "How stupid I was then, and how much smarter I'm now", and rewrite using the "more obvious and correct" variant.

...The urge to rewrite was good. But in the process you see that the "more obvious" solution is actually lacking. You even dimly remember why, because you already tried it long ago. You revert to the correct variant, but the time was wasted.

Comments that explain the solution are very important. They help to continue development the right way.

Any subtle features of the code? Where they are used?

If the code has anything subtle and counter-intuitive, it's definitely worth commenting.

Summary An important sign of a good developer is comments: their presence and even their absence.

Good comments allow us to maintain the code well, come back to it after a delay and use it more effectively.

Comment this:

Overall architecture, high-level view.
Function usage.
Important solutions, especially when not immediately obvious.

Avoid comments:

That tell “how code works” and “what it does”.
Put them in only if it’s impossible to make the code so simple and self-descriptive that it doesn’t require them.

Comments are also used for auto-documenting tools like JSDoc3: they read them and generate HTML-docs (or docs in another format).
Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, {“url”:“https://javascript.info/comments”,“identifier”:“Vcomments”}); };var DisqusShortname = “javascriptinfo”;var DisqusEnabled = true;
TutorialThe JavaScript languageCode quality{“@context”:“https://schema.org”,“@type”:“BreadcrumbList”,“itemListElement”:
[{“@type”:“ListItem”,“position”:1,“name”:“Tutorial”,“item”:“https://javascript.info/”},
{“@type”:“ListItem”,“position”:2,“name”:“The JavaScript language”,“item”:“https://javascript.info/js”},{“@type”:“ListItem”,“position”:3,“name”:“Code quality”,“item”:“https://javascript.info/code-quality”}]}September 5, 2020Ninja codeLearning without thought is labor lost; thought without learning is perilous.

Confucius (Analects)

Programmer ninjas of the past used these tricks to sharpen the mind of code maintainers.

Code review gurus look for them in test tasks.

Novice developers sometimes use them even better than programmer ninjas.

Read them carefully and find out who you are – a ninja, a novice, or maybe a code reviewer?

Irony detected

Many try to follow ninja paths. Few succeed.

Brevity is the soul of witMake the code as short as possible. Show how smart you are.

Let subtle language features guide you.

For instance, take a look at this ternary operator '?:'

```
// taken from a well-known javascript library  
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Cool, right? If you write like that, a developer who comes across this line and tries to understand what is the value of `i` is going to have a merry time. Then come to you, seeking for an answer.

Tell them that shorter is always better. Initiate them into the paths of ninja.

One-letter variablesThe Dao hides in wordlessness. Only the Dao is well begun and well completed.

Laozi (Tao Te Ching)

Another way to code shorter is to use single-letter variable names everywhere. Like `a`, `b` or `c`.

A short variable disappears in the code like a real ninja in the forest. No one will be able to find it using “search” of the editor. And even if someone does, they won’t be able to “decipher” what the name `a` or `b` means.

...But there’s an exception. A real ninja will never use `i` as the counter in a “for” loop.

Anywhere, but not here. Look around, there are many more exotic letters. For instance, `x` or `y`.

An exotic variable as a loop counter is especially cool if the loop body takes 1-2 pages (make it longer if you can). Then if someone looks deep inside the loop, they won’t be able to quickly figure out that the variable named `x` is the loop counter.

Use abbreviationsIf the team rules forbid the use of one-letter and vague names – shorten them, make abbreviations.

Like this:

```
list !' lst.
```

```
userAgent !' ua.
```

```
browser !' brsr.
```

```
...etc
```

Only the one with truly good intuition will be able to understand such names. Try to shorten everything. Only a worthy person should be able to uphold the development of your code.

Soar high. Be abstract.The great square is cornerless

The great vessel is last complete,

The great note is rarified sound,

The great image has no form.

Laozi (Tao Te Ching)

While choosing a name try to use the most abstract word. Like `obj`, `data`, `value`, `item`, `elem` and so on.

The ideal name for a variable is data. Use it everywhere you can. Indeed, every variable holds data, right?

...But what to do if data is already taken? Try value, it's also universal. After all, a variable eventually gets a value.

Name a variable by its type: str, num...

Give them a try. A young initiate may wonder – are such names really useful for a ninja? Indeed, they are!

Sure, the variable name still means something. It says what's inside the variable: a string, a number or something else. But when an outsider tries to understand the code, they'll be surprised to see that there's actually no information at all! And will ultimately fail to alter your well-thought code.

The value type is easy to find out by debugging. But what's the meaning of the variable? Which string/number does it store?

There's just no way to figure out without a good meditation!

...But what if there are no more such names? Just add a number: data1, item2, elem5...

Attention test Only a truly attentive programmer should be able to understand your code. But how to check that?

One of the ways – use similar variable names, like date and data.

Mix them where you can.

A quick read of such code becomes impossible. And when there's a typo... Ummm...

We're stuck for long, time to drink tea.

Smart synonyms The Tao that can be told is not the eternal Tao. The name that can be named is not the eternal name.

Laozi (Tao Te Ching)

Using similar names for same things makes life more interesting and shows your creativity to the public.

For instance, consider function prefixes. If a function shows a message on the screen – start it with display..., like displayMessage. And then if another function shows on the screen something else, like a user name, start it with show... (like showName).

Insinuate that there's a subtle difference between such functions, while there is none.

Make a pact with fellow ninjas of the team: if John starts "showing" functions with display... in his code, then Peter could use render..., and Ann – paint.... Note how much more interesting and diverse the code became.

...And now the hat trick!

For two functions with important differences – use the same prefix!

For instance, the function printPage(page) will use a printer. And the function printText(text) will put the text on-screen. Let an unfamiliar reader think well over similarly named function printMessage: "Where does it put the message? To a printer

or on the screen?”. To make it really shine, `printMessage(message)` should output it in the new window!

Reuse names Once the whole is divided, the parts need names.

There are already enough names.

One must know when to stop.

Laozi (Tao Te Ching)

Add a new variable only when absolutely necessary.

Instead, reuse existing names. Just write new values into them.

In a function try to use only variables passed as parameters.

That would make it really hard to identify what’s exactly in the variable now. And also where it comes from. The purpose is to develop the intuition and memory of a person reading the code. A person with weak intuition would have to analyze the code line-by-line and track the changes through every code branch.

An advanced variant of the approach is to covertly (!) replace the value with something alike in the middle of a loop or a function.

For instance:

```
function ninjaFunction(elem) {  
  // 20 lines of code working with elem  
  
  elem = clone(elem);  
  
  // 20 more lines, now working with the clone of the elem!  
}
```

A fellow programmer who wants to work with `elem` in the second half of the function will be surprised... Only during the debugging, after examining the code they will find out that they’re working with a clone!

Seen in code regularly. Deadly effective even against an experienced ninja.

Underscores for fun Put underscores `_` and `__` before variable names. Like `_name` or `__value`. It would be great if only you knew their meaning. Or, better, add them just for fun, without particular meaning at all. Or different meanings in different places.

You kill two rabbits with one shot. First, the code becomes longer and less readable, and the second, a fellow developer may spend a long time trying to figure out what the underscores mean.

A smart ninja puts underscores at one spot of code and evades them at other places. That makes the code even more fragile and increases the probability of future errors.

Show your love Let everyone see how magnificent your entities are! Names like `superElement`, `megaFrame` and `niceltem` will definitely enlighten a reader.

Indeed, from one hand, something is written: super.., mega.., nice.. But from the other hand – that brings no details. A reader may decide to look for a hidden meaning and meditate for an hour or two of their paid working time.

Overlap outer variables When in the light, can't see anything in the darkness.

When in the darkness, can see everything in the light.

Guan Yin Zi

Use same names for variables inside and outside a function. As simple. No efforts to invent new names.

```
    let user = authenticateUser();

function render() {
  let user = anotherValue();
  ...
  ...many lines...
  ...
  ... // <-- a programmer wants to work with user here and...
  ...
}
```

A programmer who jumps inside the render will probably fail to notice that there's a local user shadowing the outer one.

Then they'll try to work with user assuming that it's the external variable, the result of authenticateUser()... The trap is sprung! Hello, debugger...

Side-effects everywhere! There are functions that look like they don't change anything. Like isReady(), checkPermission(), findTags()... They are assumed to carry out calculations, find and return the data, without changing anything outside of them. In other words, without "side-effects".

A really beautiful trick is to add a "useful" action to them, besides the main task.

An expression of dazed surprise on the face of your colleague when they see a function named is.., check.. or find... changing something – will definitely broaden your boundaries of reason.

Another way to surprise is to return a non-standard result.

Show your original thinking! Let the call of checkPermission return not true/false, but a complex object with the results of the check.

Those developers who try to write if (checkPermission(..)), will wonder why it doesn't work. Tell them: "Read the docs!". And give this article.

Powerful functions! The great Tao flows everywhere, both to the left and to the right.

Laozi (Tao Te Ching)

Don't limit the function by what's written in its name. Be broader.

For instance, a function `validateEmail(email)` could (besides checking the email for correctness) show an error message and ask to re-enter the email.

Additional actions should not be obvious from the function name. A true ninja coder will make them not obvious from the code as well.

Joining several actions into one protects your code from reuse.

Imagine, another developer wants only to check the email, and not output any message. Your function `validateEmail(email)` that does both will not suit them. So they won't break your meditation by asking anything about it.

SummaryAll "pieces of advice" above are from the real code... Sometimes, written by experienced developers. Maybe even more experienced than you are ;)

Follow some of them, and your code will become full of surprises.

Follow many of them, and your code will become truly yours, no one would want to change it.

Follow all, and your code will become a valuable lesson for young developers looking for enlightenment.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var disqus_config = function() { if (!this.page) this.page = {};  
Object.assign(this.page, {"url":"https://javascript.info/ninja-code","identifier":"Vninja-code"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;
```

```
TutorialThe JavaScript languageCode quality{"@context":"https://  
schema.org","@type":"BreadcrumbList","itemListElement":  
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},  
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Code quality","item":"https://  
javascript.info/code-quality"}]}October 14, 2022Automated testing with
```

MochaAutomated testing will be used in further tasks, and it's also widely used in real projects.

Why do we need tests?When we write a function, we can usually imagine what it should do: which parameters give which results.

During development, we can check the function by running it and comparing the outcome with the expected one. For instance, we can do it in the console.

If something is wrong – then we fix the code, run again, check the result – and so on till it works.

But such manual "re-runs" are imperfect.

When testing a code by manual re-runs, it's easy to miss something.

For instance, we're creating a function `f`. Wrote some code, testing: `f(1)` works, but `f(2)` doesn't work. We fix the code and now `f(2)` works. Looks complete? But we forgot to re-test `f(1)`. That may lead to an error.

That's very typical. When we develop something, we keep a lot of possible use cases in mind. But it's hard to expect a programmer to check all of them manually after every change. So it becomes easy to fix one thing and break another one.

Automated testing means that tests are written separately, in addition to the code. They run our functions in various ways and compare results with the expected.

Behavior Driven Development (BDD) Let's start with a technique named Behavior Driven Development or, in short, BDD.

BDD is three things in one: tests AND documentation AND examples.

To understand BDD, we'll examine a practical case of development.

Development of "pow": the spec Let's say we want to make a function `pow(x, n)` that raises `x` to an integer power `n`. We assume that `n ≥ 0`.

That task is just an example: there's the `**` operator in JavaScript that can do that, but here we concentrate on the development flow that can be applied to more complex tasks as well.

Before creating the code of `pow`, we can imagine what the function should do and describe it.

Such description is called a specification or, in short, a spec, and contains descriptions of use cases together with tests for them, like this:

```
describe("pow", function() {  
  
  it("raises to n-th power", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
});
```

A spec has three main building blocks that you can see above:

```
describe("title", function() { ... })
```

What functionality we're describing? In our case we're describing the function `pow`. Used to group "workers" – the `it` blocks.

```
it("use case description", function() { ... })
```

In the title of `it` we in a human-readable way describe the particular use case, and the second argument is a function that tests it.

```
assert.equal(value1, value2)
```

The code inside it block, if the implementation is correct, should execute without errors. Functions `assert.*` are used to check whether `pow` works as expected. Right here we're using one of them – `assert.equal`, it compares arguments and yields an error if they are not equal. Here it checks that the result of `pow(2, 3)` equals 8. There are other types of comparisons and checks, that we'll add later.

The specification can be executed, and it will run the test specified in it block. We'll see that later.

The development flowThe flow of development usually looks like this:

An initial spec is written, with tests for the most basic functionality.

An initial implementation is created.

To check whether it works, we run the testing framework Mocha (more details soon) that runs the spec. While the functionality is not complete, errors are displayed. We make corrections until everything works.

Now we have a working initial implementation with tests.

We add more use cases to the spec, probably not yet supported by the implementations. Tests start to fail.

Go to 3, update the implementation till tests give no errors.

Repeat steps 3-6 till the functionality is ready.

So, the development is iterative. We write the spec, implement it, make sure tests pass, then write more tests, make sure they work etc. At the end we have both a working implementation and tests for it.

Let's see this development flow in our practical case.

The first step is already complete: we have an initial spec for `pow`. Now, before making the implementation, let's use a few JavaScript libraries to run the tests, just to see that they are working (they will all fail).

The spec in actionHere in the tutorial we'll be using the following JavaScript libraries for tests:

Mocha – the core framework: it provides common testing functions including `describe` and `it` and the main function that runs tests.

Chai – the library with many assertions. It allows to use a lot of different assertions, for now we need only `assert.equal`.

Sinon – a library to spy over functions, emulate built-in functions and more, we'll need it much later.

These libraries are suitable for both in-browser and server-side testing. Here we'll consider the browser variant.

The full HTML page with these frameworks and `pow` spec:


```

    <!DOCTYPE html>
<html>
<head>
  <!-- add mocha css, to show results -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/
mocha.css">
  <!-- add mocha framework code -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
  <script>
    mocha.setup('bdd'); // minimal setup
  </script>
  <!-- add chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
  <script>
    // chai has a lot of stuff, let's make assert global
    let assert = chai.assert;
  </script>
</head>

<body>

  <script>
    function pow(x, n) {
      /* function code is to be written, empty now */
    }
  </script>

  <!-- the script with tests (describe, it...) -->
  <script src="test.js"></script>

  <!-- the element with id="mocha" will contain test results -->
  <div id="mocha"></div>

  <!-- run tests! -->
  <script>
    mocha.run();
  </script>
</body>

</html>

```

The page can be divided into five parts:

The <head> – add third-party libraries and styles for tests.

The `<script>` with the function to test, in our case – with the code for `pow`.
The tests – in our case an external script `test.js` that has `describe("pow", ...)` from above.
The HTML element `<div id="mocha">` will be used by Mocha to output results.
The tests are started by the command `mocha.run()`.

The result:

As of now, the test fails, there's an error. That's logical: we have an empty function code in `pow`, so `pow(2,3)` returns `undefined` instead of `8`.
For the future, let's note that there are more high-level test-runners, like `karma` and others, that make it easy to autorun many different tests.
Initial implementationLet's make a simple implementation of `pow`, for tests to pass:

```
function pow(x, n) {  
  return 8; // :) we cheat!  
}
```

Wow, now it works!

Improving the specWhat we've done is definitely a cheat. The function does not work: an attempt to calculate `pow(3,4)` would give an incorrect result, but tests pass.
...But the situation is quite typical, it happens in practice. Tests pass, but the function works wrong. Our spec is imperfect. We need to add more use cases to it.
Let's add one more test to check that `pow(3, 4) = 81`.
We can select one of two ways to organize the test here:

The first variant – add one more assert into the same it:

```
describe("pow", function() {  
  
  it("raises to n-th power", function() {  
    assert.equal(pow(2, 3), 8);  
    assert.equal(pow(3, 4), 81);  
  });  
  
});
```

The second – make two tests:

```
describe("pow", function() {  
  
  it("2 raised to power 3 is 8", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
  it("3 raised to power 4 is 81", function() {  
    assert.equal(pow(3, 4), 81);  
  });  
  
});
```

The principal difference is that when assert triggers an error, the it block immediately terminates. So, in the first variant if the first assert fails, then we'll never see the result of the second assert.

Making tests separate is useful to get more information about what's going on, so the second variant is better.

And besides that, there's one more rule that's good to follow.

One test checks one thing.

If we look at the test and see two independent checks in it, it's better to split it into two simpler ones.

So let's continue with the second variant.

The result:

As we could expect, the second test failed. Sure, our function always returns 8, while the assert expects 81.

Improving the implementationLet's write something more real for tests to pass:

```
function pow(x, n) {  
  let result = 1;  
  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  
  return result;  
}
```

To be sure that the function works well, let's test it for more values. Instead of writing it blocks manually, we can generate them in for:

```
describe("pow", function() {  
  
  function makeTest(x) {  
    let expected = x * x * x;  
    it(`${x} in the power 3 is ${expected}`, function() {  
      assert.equal(pow(x, 3), expected);  
    });  
  }  
  
  for (let x = 1; x <= 5; x++) {  
    makeTest(x);  
  }  
  
});
```

The result:

Nested describeWe're going to add even more tests. But before that let's note that the helper function `makeTest` and `for` should be grouped together. We won't need `makeTest` in other tests, it's needed only in `for`: their common task is to check how `pow` raises into the given power.

Grouping is done with a nested `describe`:

```
describe("pow", function() {  
  
  describe("raises x to power 3", function() {  
  
    function makeTest(x) {  
      let expected = x * x * x;  
      it(`${x} in the power 3 is ${expected}`, function() {  
        assert.equal(pow(x, 3), expected);  
      });  
    }  
  
    for (let x = 1; x <= 5; x++) {  
      makeTest(x);  
    }  
  
  });  
  
  // ... more tests to follow here, both describe and it can be added  
});
```

The nested `describe` defines a new “subgroup” of tests. In the output we can see the titled indentation:

In the future we can add more it and describe on the top level with helper functions of their own, they won't see makeTest.

before/after and beforeEach/afterEach

We can setup before/after functions that execute before/after running tests, and also beforeEach/afterEach functions that execute before/after every it. For instance:

```
describe("test", function() {  
  
  before(() => alert("Testing started – before all tests"));  
  after(() => alert("Testing finished – after all tests"));  
  
  beforeEach(() => alert("Before a test – enter a test"));  
  afterEach(() => alert("After a test – exit a test"));  
  
  it('test 1', () => alert(1));  
  it('test 2', () => alert(2));  
  
});
```

The running sequence will be:

```
Testing started – before all tests (before)  
Before a test – enter a test (beforeEach)  
1  
After a test – exit a test (afterEach)  
Before a test – enter a test (beforeEach)  
2  
After a test – exit a test (afterEach)  
Testing finished – after all tests (after)
```

Open the example in the sandbox. Usually, beforeEach/afterEach and before/after are used to perform initialization, zero out counters or do something else between the

tests (or test groups).

Extending the specThe basic functionality of `pow` is complete. The first iteration of the development is done. When we're done celebrating and drinking champagne – let's go on and improve it.

As it was said, the function `pow(x, n)` is meant to work with positive integer values `n`. To indicate a mathematical error, JavaScript functions usually return `NaN`. Let's do the same for invalid values of `n`.

Let's first add the behavior to the spec(!):

```
describe("pow", function() {  
  
  // ...  
  
  it("for negative n the result is NaN", function() {  
    assert.isNaN(pow(2, -1));  
  });  
  
  it("for non-integer n the result is NaN", function() {  
    assert.isNaN(pow(2, 1.5));  
  });  
  
});
```

The result with new tests:

The newly added tests fail, because our implementation does not support them. That's how BDD is done: first we write failing tests, and then make an implementation for them.

Other assertions

Please note the assertion `assert.isNaN`: it checks for `NaN`.

There are other assertions in Chai as well, for instance:

`assert.equal(value1, value2)` – checks the equality `value1 == value2`.

`assert.strictEqual(value1, value2)` – checks the strict equality `value1 === value2`.

`assert.notEqual`, `assert.notStrictEqual` – inverse checks to the ones above.

assert.isTrue(value) – checks that value === true
assert.isFalse(value) – checks that value === false
...the full list is in the docs

So we should add a couple of lines to pow:

```
function pow(x, n) {  
  if (n < 0) return NaN;  
  if (Math.round(n) !== n) return NaN;  
  
  let result = 1;  
  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  
  return result;  
}
```

Now it works, all tests pass:

Open the full final example in the sandbox. Summary In BDD, the spec goes first, followed by implementation. At the end we have both the spec and the code.

The spec can be used in three ways:

As Tests – they guarantee that the code works correctly.

As Docs – the titles of describe and it tell what the function does.

As Examples – the tests are actually working examples showing how a function can be used.

With the spec, we can safely improve, change, even rewrite the function from scratch and make sure it still works right.

That's especially important in large projects when a function is used in many places.

When we change such a function, there's just no way to manually check if every place that uses it still works right.

Without tests, people have two ways:

To perform the change, no matter what. And then our users meet bugs, as we probably fail to check something manually.

Or, if the punishment for errors is harsh, as there are no tests, people become afraid to modify such functions, and then the code becomes outdated, no one wants to get into it. Not good for development.

Automatic testing helps to avoid these problems!

If the project is covered with tests, there's just no such problem. After any changes, we can run tests and see a lot of checks made in a matter of seconds.

Besides, a well-tested code has better architecture.

Naturally, that's because auto-tested code is easier to modify and improve. But there's also another reason.

To write tests, the code should be organized in such a way that every function has a clearly described task, well-defined input and output. That means a good architecture from the beginning.

In real life that's sometimes not that easy. Sometimes it's difficult to write a spec before the actual code, because it's not yet clear how it should behave. But in general writing tests makes development faster and more stable.

Later in the tutorial you will meet many tasks with tests baked-in. So you'll see more practical examples.

Writing tests requires good JavaScript knowledge. But we're just starting to learn it. So, to settle down everything, as of now you're not required to write tests, but you should already be able to read them even if they are a little bit more complex than in this chapter.

TasksWhat's wrong in the test?importance: 5What's wrong in the test of pow below?

```
it("Raises x to the power n", function() {
  let x = 5;

  let result = x;
  assert.equal(pow(x, 1), result);

  result *= x;
  assert.equal(pow(x, 2), result);

  result *= x;
  assert.equal(pow(x, 3), result);
});
```

P.S. Syntactically the test is correct and passes.
solutionThe test demonstrates one of the temptations a developer meets when writing tests.
What we have here is actually 3 tests, but layed out as a single function with 3 asserts. Sometimes it's easier to write this way, but if an error occurs, it's much less obvious what went wrong.
If an error happens in the middle of a complex execution flow, then we'll have to figure out the data at that point. We'll actually have to debug the test.
It would be much better to break the test into multiple it blocks with clearly written inputs and outputs.
Like this:

```
describe("Raises x to power n", function() {  
  it("5 in the power of 1 equals 5", function() {  
    assert.equal(pow(5, 1), 5);  
  });  
  
  it("5 in the power of 2 equals 25", function() {  
    assert.equal(pow(5, 2), 25);  
  });  
  
  it("5 in the power of 3 equals 125", function() {  
    assert.equal(pow(5, 3), 125);  
  });  
});
```

We replaced the single it with describe and a group of it blocks. Now if something fails we would see clearly what the data was.
Also we can isolate a single test and run it in standalone mode by writing it.only instead of it:

```
describe("Raises x to power n", function() {  
  it("5 in the power of 1 equals 5", function() {  
    assert.equal(pow(5, 1), 5);  
  });  
  
  // Mocha will run only this block  
  it.only("5 in the power of 2 equals 25", function() {
```

```

    assert.equal(pow(5, 2), 25);
  });

  it("5 in the power of 3 equals 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/testing-mocha", "identifier": "Vtesting-mocha" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageCode quality{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Code quality", "item": "https://javascript.info/code-quality"}]}October 14, 2022Polyfills and transpilersThe JavaScript language steadily evolves. New proposals to the language appear regularly, they are analyzed and, if considered worthy, are appended to the list at <https://tc39.github.io/ecma262/> and then progress to the specification. Teams behind JavaScript engines have their own ideas about what to implement first. They may decide to implement proposals that are in draft and postpone things that are already in the spec, because they are less interesting or just harder to do. So it's quite common for an engine to implement only part of the standard. A good page to see the current state of support for language features is <https://kangax.github.io/compat-table/es6/> (it's big, we have a lot to study yet). As programmers, we'd like to use most recent features. The more good stuff – the better! On the other hand, how to make our modern code work on older engines that don't understand recent features yet? There are two tools for that:

Transpilers.
Polyfills.

Here, in this chapter, our purpose is to get the gist of how they work, and their place in web development.

TranspilersA transpiler is a special piece of software that translates source code to another source code. It can parse ("read and understand") modern code and rewrite it

using older syntax constructs, so that it'll also work in outdated engines.

E.g. JavaScript before year 2020 didn't have the "nullish coalescing operator" `??`. So, if a visitor uses an outdated browser, it may fail to understand the code like `height = height ?? 100`.

A transpiler would analyze our code and rewrite `height ?? 100` into `(height !== undefined && height !== null) ? height : 100`.

```
// before running the transpiler  
height = height ?? 100;
```

```
// after running the transpiler  
height = (height !== undefined && height !== null) ? height : 100;
```

Now the rewritten code is suitable for older JavaScript engines.

Usually, a developer runs the transpiler on their own computer, and then deploys the transpiled code to the server.

Speaking of names, Babel is one of the most prominent transpilers out there.

Modern project build systems, such as webpack, provide a means to run a transpiler automatically on every code change, so it's very easy to integrate into the development process.

Polyfills New language features may include not only syntax constructs and operators, but also built-in functions.

For example, `Math.trunc(n)` is a function that "cuts off" the decimal part of a number, e.g. `Math.trunc(1.23)` returns 1.

In some (very outdated) JavaScript engines, there's no `Math.trunc`, so such code will fail.

As we're talking about new functions, not syntax changes, there's no need to transpile anything here. We just need to declare the missing function.

A script that updates/adds new functions is called "polyfill". It "fills in" the gap and adds missing implementations.

For this particular case, the polyfill for `Math.trunc` is a script that implements it, like this:

```
if (!Math.trunc) { // if no such function  
  // implement it  
  Math.trunc = function(number) {  
    // Math.ceil and Math.floor exist even in ancient JavaScript engines  
    // they are covered later in the tutorial  
    return number < 0 ? Math.ceil(number) : Math.floor(number);  
  }  
}
```

```
};  
}
```

JavaScript is a highly dynamic language. Scripts may add/modify any function, even built-in ones.

Two interesting polyfill libraries are:

core.js that supports a lot, allows to include only needed features.

polyfill.io service that provides a script with polyfills, depending on the features and user's browser.

Summary In this chapter we'd like to motivate you to study modern and even "bleeding-edge" language features, even if they aren't yet well-supported by JavaScript engines. Just don't forget to use a transpiler (if using modern syntax or operators) and polyfills (to add functions that may be missing). They'll ensure that the code works.

For example, later when you're familiar with JavaScript, you can setup a code build system based on webpack with the babel-loader plugin.

Good resources that show the current state of support for various features:

<https://kangax.github.io/compat-table/es6/> – for pure JavaScript.

<https://caniuse.com/> – for browser-related functions.

P.S. Google Chrome is usually the most up-to-date with language features, try it if a tutorial demo fails. Most tutorial demos work with any modern browser though.

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/polyfills", "identifier": "V  
polyfills" }); };  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;
```

TutorialThe JavaScript language{"@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}]
```

Objects: the basicsObjectsObject references and copyingGarbage collectionObject methods, "this"Constructor, operator "new"Optional chaining

'?.'Symbol typeObject to primitive conversionCtrl + !•Previous lessonCtrl + !•Next lessonShareTutorial map

TutorialThe JavaScript languageObjects: the basics{"@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://
```

```
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Objects: the basics", "item": "https://javascript.info/object-basics"}]]
```

June 19, 2022

Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets `{...}` with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It’s easy to find a file by its name or add/remove a file.

An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {}; // "object literal" syntax
```

Usually, the figure brackets `{...}` are used. That declaration is called an object literal. Literals and properties

We can immediately put some properties into `{...}` as “key: value” pairs:

```
let user = { // an object
  name: "John", // by key "name" store value "John"
  age: 30 // by key "age" store value 30
};
```

A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it.

In the user object, there are two properties:

The first property has the name "name" and the value "John".

The second one has the name "age" and the value 30.

The resulting user object can be imagined as a cabinet with two signed files labeled “name” and “age”.

We can add, remove and read files from it at any time.
Property values are accessible using the dot notation:

```
// get property values of the object:  
alert( user.name ); // John  
alert( user.age ); // 30
```

The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```

To remove a property, we can use the delete operator:

```
delete user.age;
```

We can also use multiword property names, but then they must be quoted:

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true // multiword property name must be quoted  
};
```

The last property in the list may end with a comma:

```
let user = {  
  name: "John",  
  age: 30,  
}
```

That is called a “trailing” or “hanging” comma. Makes it easier to add/remove/move around properties, because all lines become alike.

Square brackets For multiword properties, the dot access doesn't work:


```
// this would give a syntax error
user.likes birds = true
```

JavaScript doesn't understand that. It thinks that we address `user.likes`, and then gives a syntax error when comes across unexpected `birds`. The dot requires the key to be a valid variable identifier. That implies: contains no spaces, doesn't start with a digit and doesn't include special characters (`$` and `_` are allowed).

There's an alternative "square bracket notation" that works with any string:

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Now everything is fine. Please note that the string inside the brackets is properly quoted (any type of quotes will do).

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
let key = "likes birds";  
  
// same as user["likes birds"] = true;  
user[key] = true;
```

Here, the variable key may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility. For instance:

```
let user = {  
  name: "John",  
  age: 30  
};  
  
let key = prompt("What do you want to know about the user?", "name");  
  
// access by variable  
alert( user[key] ); // John (if enter "name")
```

The dot notation cannot be used in a similar way:

```
    let user = {  
      name: "John",  
      age: 30  
    };  
  
    let key = "name";  
    alert( user.key ) // undefined
```

Computed properties We can use square brackets in an object literal, when creating an object. That's called computed properties.
For instance:

```
    let fruit = prompt("Which fruit to buy?", "apple");  
  
    let bag = {  
      [fruit]: 5, // the name of the property is taken from the variable fruit  
    };  
  
    alert( bag.apple ); // 5 if fruit="apple"
```

The meaning of a computed property is simple: [fruit] means that the property name should be taken from fruit.
So, if a visitor enters "apple", bag will become {apple: 5}.
Essentially, that works the same as:

```
    let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// take property name from the fruit variable
bag[fruit] = 5;
```

...But looks nicer.
We can use more complex expressions inside square brackets:

```
    let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Square brackets are much more powerful than dot notation. They allow any property names and variables. But they are also more cumbersome to write. So most of the time, when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets. Property value shorthand
In real code, we often use existing variables as values for property names.
For instance:

```
function makeUser(name, age) {  
  return {  
    name: name,  
    age: age,  
    // ...other properties  
  };  
}  
  
let user = makeUser("John", 30);  
alert(user.name); // John
```

In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there's a special property value shorthand to make it shorter. Instead of `name:name` we can just write `name`, like this:

```
function makeUser(name, age) {  
  return {  
    name, // same as name: name  
    age, // same as age: age  
    // ...  
  };  
}
```

We can use both normal properties and shorthands in the same object:

```
let user = {  
  name, // same as name:name  
  age: 30  
};
```

Property names limitationsAs we already know, a variable cannot have a name equal to one of the language-reserved words like “for”, “let”, “return” etc. But for an object property, there’s no such restriction:

```
// these properties are all right
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return ); // 6
```

In short, there are no limitations on property names. They can be any strings or symbols (a special type for identifiers, to be covered later). Other types are automatically converted to strings. For instance, a number 0 becomes a string "0" when used as a property key:

```
let obj = {
```

```
0: "test" // same as "0": "test"
};
```

```
// both alerts access the same property (the number 0 is converted to string "0")
alert( obj["0"] ); // test
alert( obj[0] ); // test (same property)
```

There's a minor gotcha with a special property named `__proto__`. We can't set it to a non-object value:

```
let obj = {};
obj.__proto__ = 5; // assign a number
alert(obj.__proto__); // [object Object] - the value is an object, didn't work as intended
```

As we see from the code, the assignment to a primitive 5 is ignored. We'll cover the special nature of `__proto__` in subsequent chapters, and suggest the ways to fix such behavior.

Property existence test, "in" operator A notable feature of objects in JavaScript, compared to many other languages, is that it's possible to access any property. There will be no error if the property doesn't exist!

Reading a non-existing property just returns undefined. So we can easily test whether the property exists:

```
let user = {};  
  
alert( user.noSuchProperty === undefined ); // true means "no such property"
```

There's also a special operator "in" for that.
The syntax is:

"key" in object

For instance:

```
let user = { name: "John", age: 30 };  
  
alert( "age" in user ); // true, user.age exists  
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Please note that on the left side of in there must be a property name. That's usually a quoted string.
If we omit quotes, that means a variable should contain the actual name to be tested.
For instance:


```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, property "age" exists
```

Why does the in operator exist? Isn't it enough to compare against undefined? Well, most of the time the comparison with undefined works fine. But there's a special case when it fails, but "in" works correctly. It's when an object property exists, but stores undefined:

```
let obj = {
  test: undefined
};

alert( obj.test ); // it's undefined, so - no such property?

alert( "test" in obj ); // true, the property does exist!
```

In the code above, the property obj.test technically exists. So the in operator works right. Situations like this happen very rarely, because undefined should not be explicitly assigned. We mostly use null for “unknown” or “empty” values. So the in operator is an

exotic guest in the code.

The "for..in" loop To walk over all keys of an object, there exists a special form of the loop: for..in. This is a completely different thing from the for(;;) construct that we studied before.

The syntax:

```
    for (key in object) {  
  // executes the body for each key among object properties  
}
```

For instance, let's output all properties of user:

```
    let user = {  
  name: "John",  
  age: 30,  
  isAdmin: true  
};  
  
for (let key in user) {  
  // keys  
  alert( key ); // name, age, isAdmin  
  // values for the keys  
  alert( user[key] ); // John, 30, true  
}
```

Note that all "for" constructs allow us to declare the looping variable inside the loop, like let key here.

Also, we could use another variable name here instead of key. For instance, "for (let

prop in obj)" is also widely used.

Ordered like an objectAre objects ordered? In other words, if we loop over an object, do we get all properties in the same order they were added? Can we rely on this?

The short answer is: “ordered in a special fashion”: integer properties are sorted, others appear in creation order. The details follow.

As an example, let’s consider an object with the phone codes:

```
let codes = {  
  "49": "Germany",  
  "41": "Switzerland",  
  "44": "Great Britain",  
  // ..,  
  "1": "USA"  
};  
  
for (let code in codes) {  
  alert(code); // 1, 41, 44, 49  
}
```

The object may be used to suggest a list of options to the user. If we’re making a site mainly for a German audience then we probably want 49 to be the first.

But if we run the code, we see a totally different picture:

USA (1) goes first
then Switzerland (41) and so on.

The phone codes go in the ascending sorted order, because they are integers. So we see 1, 41, 44, 49.

Integer properties? What’s that?

The “integer property” term here means a string that can be converted to-and-from an integer without a change.

So, "49" is an integer property name, because when it’s transformed to an integer

number and back, it's still the same. But "+49" and "1.2" are not:

```
// Number(...) explicitly converts to a number
// Math.trunc is a built-in function that removes the decimal part
alert( String(Math.trunc(Number("49"))) ); // "49", same, integer property
alert( String(Math.trunc(Number("+49"))) ); // "49", not same "+49" !Ò not integer property
alert( String(Math.trunc(Number("1.2"))) ); // "1", not same "1.2" !Ò not integer property
```

...On the other hand, if the keys are non-integer, then they are listed in the creation order, for instance:

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // add one more

// non-integer properties are listed in the creation order
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

So, to fix the issue with the phone codes, we can “cheat” by making the codes non-integer. Adding a plus "+" sign before each code is enough.
Like this:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Now it works as intended.
SummaryObjects are associative arrays with several special features.
They store properties (key-value pairs), where:

Property keys must be strings or symbols (usually strings).
Values can be of any type.

To access a property, we can use:

The dot notation: `obj.property`.
Square brackets notation `obj["property"]`. Square brackets allow taking the key from a variable, like `obj[varWithKey]`.

Additional operators:

To delete a property: delete obj.prop.
To check if a property with the given key exists: "key" in obj.
To iterate over an object: for (let key in obj) loop.

What we've studied in this chapter is called a "plain object", or just Object.
There are many other kinds of objects in JavaScript:

Array to store ordered data collections,
Date to store the information about the date and time,
Error to store the information about an error.
...And so on.

They have their special features that we'll study later. Sometimes people say something like "Array type" or "Date type", but formally they are not types of their own, but belong to a single "object" data type. And they extend it in various ways.
Objects in JavaScript are very powerful. Here we've just scratched the surface of a topic that is really huge. We'll be closely working with objects and learning more about them in further parts of the tutorial.
TasksHello, objectimportance: 5Write the code, one line for each action:

Create an empty object user.
Add the property name with the value John.
Add the property surname with the value Smith.
Change the value of the name to Pete.
Remove the property name from the object.

solution

```
let user = {};  
user.name = "John";  
user.surname = "Smith";  
user.name = "Pete";  
delete user.name;
```

Check for emptinessimportance: 5Write the function isEmpty(obj) which returns true if the object has no properties, false otherwise.
Should work like that:

```
let schedule = {};  
  
alert( isEmpty(schedule) ); // true  
  
schedule["8:30"] = "get up";  
  
alert( isEmpty(schedule) ); // false
```

Open a sandbox with tests. solution Just loop over the object and return false immediately if there's at least one property.

```
function isEmpty(obj) {  
  for (let key in obj) {  
    // if the loop has started, there is a property  
    return false;  
  }  
  return true;  
}
```

Open the solution with tests in a sandbox. Sum object properties importance: 5 We have an object storing salaries of our team:

```
let salaries = {  
  John: 100,  
  Ann: 160,  
  Pete: 130  
}
```

Write the code to sum all salaries and store in the variable sum. Should be 390 in the example above.
If salaries is empty, then the result must be 0.
solution

```
    let salaries = {  
      John: 100,  
      Ann: 160,  
      Pete: 130  
    };  
  
    let sum = 0;  
    for (let key in salaries) {  
      sum += salaries[key];  
    }  
  
    alert(sum); // 390
```

Multiply numeric property values by 2importance: 3Create a function multiplyNumeric(obj) that multiplies all numeric property values of obj by 2. For instance:

```
    // before the call  
    let menu = {  
      width: 200,  
      height: 300,  
      title: "My menu"  
    };  
  
    multiplyNumeric(menu);  
  
    // after the call  
    menu = {  
      width: 400,  
      height: 600,  
      title: "My menu"  
    };
```


Please note that multiplyNumeric does not need to return anything. It should modify the object in-place.

P.S. Use typeof to check for a number here.

Open a sandbox with tests.solution

```
function multiplyNumeric(obj) {  
  for (let key in obj) {  
    if (typeof obj[key] == 'number') {  
      obj[key] *= 2;  
    }  
  }  
}
```

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info","object":{"identifier":"Object"}}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageObjects: the basics{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Objects: the basics","item":"https://javascript.info/object-basics"]}]October 1, 2022Object references and copyingOne of the fundamental differences of objects versus primitives is that objects are stored and copied “by reference”, whereas primitive values: strings, numbers, booleans, etc – are always copied “as a whole value”. That’s easy to understand if we look a bit under the hood of what happens when we copy a value.

Let’s start with a primitive, such as a string.

Here we put a copy of message into phrase:

```
let message = "Hello!";  
let phrase = message;
```

As a result we have two independent variables, each one storing the string "Hello!".

Quite an obvious result, right?

Objects are not like that.

A variable assigned to an object stores not the object itself, but its “address in memory” – in other words “a reference” to it.

Let’s look at an example of such a variable:

```
let user = {  
  name: "John"  
};
```

And here’s how it’s actually stored in memory:

The object is stored somewhere in memory (at the right of the picture), while the user variable (at the left) has a “reference” to it.

We may think of an object variable, such as user, like a sheet of paper with the address of the object on it.

When we perform actions with the object, e.g. take a property user.name, the JavaScript engine looks at what’s at that address and performs the operation on the actual object.

Now here’s why it’s important.

When an object variable is copied, the reference is copied, but the object itself is not duplicated.

For instance:

```
let user = { name: "John" };  
let admin = user; // copy the reference
```

Now we have two variables, each storing a reference to the same object:

As you can see, there's still one object, but now with two variables that reference it. We can use either variable to access the object and modify its contents:

```
let user = { name: 'John' };  
let admin = user;  
admin.name = 'Pete'; // changed by the "admin" reference  
alert(user.name); // 'Pete', changes are seen from the "user" reference
```

It's as if we had a cabinet with two keys and used one of them (admin) to get into it and make changes. Then, if we later use another key (user), we are still opening the same cabinet and can access the changed contents.

Comparison by reference Two objects are equal only if they are the same object. For instance, here a and b reference the same object, thus they are equal:

```
    let a = {};  
let b = a; // copy the reference  
  
alert( a == b ); // true, both variables reference the same object  
alert( a === b ); // true
```

And here two independent objects are not equal, even though they look alike (both are empty):

```
    let a = {};  
let b = {}; // two independent objects  
  
alert( a == b ); // false
```

For comparisons like `obj1 > obj2` or for a comparison against a primitive `obj == 5`, objects are converted to primitives. We'll study how object conversions work very soon, but to tell the truth, such comparisons are needed very rarely – usually they appear as a result of a programming mistake.

Const objects can be modified

An important side effect of storing objects as references is that an object declared as `const` can be modified.
For instance:

```
const user = {  
  name: "John"  
};  
  
user.name = "Pete"; // (*)  
  
alert(user.name); // Pete
```

It might seem that the line (*) would cause an error, but it does not. The value of `user` is constant, it must always reference the same object, but properties of that object are free to change.

In other words, the `const user` gives an error only if we try to set `user=...` as a whole. That said, if we really need to make constant object properties, it's also possible, but using totally different methods. We'll mention that in the chapter [Property flags and descriptors](#).

Cloning and merging, `Object.assign`, copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object?

We can create a new object and replicate the structure of the existing one, by iterating over its properties and copying them on the primitive level.

Like this:

```
    let user = {  
    name: "John",  
    age: 30  
};  
  
let clone = {}; // the new empty object  
  
// let's copy all user properties into it  
for (let key in user) {  
    clone[key] = user[key];  
}  
  
// now clone is a fully independent object with the same content  
clone.name = "Pete"; // changed the data in it  
  
alert( user.name ); // still John in the original object
```

We can also use the method `Object.assign`.
The syntax is:

```
Object.assign(dest, ...sources)
```

The first argument `dest` is a target object.
Further arguments is a list of source objects.

It copies the properties of all source objects into the target `dest`, and then returns it as the result.

For example, we have `user` object, let's add a couple of permissions to it:

```
let user = { name: "John" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copies all properties from permissions1 and permissions2 into user
Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }
alert(user.name); // John
alert(user.canView); // true
alert(user.canEdit); // true
```

If the copied property name already exists, it gets overwritten:

```
let user = { name: "John" };

Object.assign(user, { name: "Pete" });

alert(user.name); // now user = { name: "Pete" }
```

We also can use `Object.assign` to perform a simple object cloning:

```
    let user = {  
      name: "John",  
      age: 30  
    };  
  
    let clone = Object.assign({}, user);  
  
    alert(clone.name); // John  
    alert(clone.age); // 30
```

Here it copies all properties of user into the empty object and returns it.
There are also other methods of cloning an object, e.g. using the spread syntax `clone = {...user}`, covered later in the tutorial.
Nested cloning Until now we assumed that all properties of user are primitive. But properties can be references to other objects.
Like this:

```
    let user = {  
      name: "John",  
      sizes: {  
        height: 182,  
        width: 50  
      }  
    };  
  
    alert( user.sizes.height ); // 182
```


Now it's not enough to copy `clone.sizes = user.sizes`, because `user.sizes` is an object, and will be copied by reference, so `clone` and `user` will share the same `sizes`:

```
    let user = {
name: "John",
sizes: {
  height: 182,
  width: 50
}
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, same object

// user and clone share sizes
user.sizes.width = 60; // change a property from one place
alert(clone.sizes.width); // 60, get the result from the other one
```

To fix that and make `user` and `clone` truly separate objects, we should use a cloning loop that examines each value of `user[key]` and, if it's an object, then replicate its structure as well. That is called a “deep cloning” or “structured cloning”. There's `structuredClone` method that implements deep cloning. The call `structuredClone(object)` clones the object with all nested properties. Here's how we can use it in our example:

```
    let user = {  
name: "John",  
sizes: {  
  height: 182,  
  width: 50  
}  
};  
  
let clone = structuredClone(user);  
  
alert( user.sizes === clone.sizes ); // false, different objects  
  
// user and clone are totally unrelated now  
user.sizes.width = 60; // change a property from one place  
alert(clone.sizes.width); // 50, not related
```

The structuredClone method can clone most data types, such as objects, arrays, primitive values.
It also supports circular references, when an object property references the object itself (directly or via a chain of references).
For instance:

```
    let user = {};  
// let's create a circular reference:  
// user.me references the user itself
```

```
user.me = user;
```

```
let clone = structuredClone(user);  
alert(clone.me === clone); // true
```

As you can see, clone.me references the clone, not the user! So the circular reference was cloned correctly as well.

Although, there are cases when structuredClone fails.

For instance, when an object has a function property:

```
// error  
structuredClone({  
  f: function() {}  
});
```

Function properties aren't supported.

To handle such complex cases we may need to use a combination of cloning methods, write custom code or, to not reinvent the wheel, take an existing implementation, for instance `_.cloneDeep(obj)` from the JavaScript library `lodash`.

Summary Objects are assigned and copied by reference. In other words, a variable stores not the “object value”, but a “reference” (address in memory) for the value. So copying such a variable or passing it as a function argument copies that reference, not the object itself.

All operations via copied references (like adding/removing properties) are performed on the same single object.

To make a “real copy” (a clone) we can use `Object.assign` for the so-called “shallow copy” (nested objects are copied by reference) or a “deep cloning” function `structuredClone` or use a custom cloning implementation, such as `_.cloneDeep(obj)`.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/object-copy", "identifier": "Vobject-copy" }); };  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;  
TutorialThe JavaScript languageObjects: the basics{"@context": "https://  
schema.org", "@type": "BreadcrumbList", "itemListElement":  
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Objects: the  
basics", "item": "https://javascript.info/object-basics"}]}
```

October 14, 2022

Garbage collection

Memory management in JavaScript is performed automatically and invisibly to us. We create primitives, objects, functions... All that takes memory.

What happens when something is not needed any more? How does the JavaScript engine discover it and clean it up?

Reachability

The main concept of memory management in JavaScript is reachability. Simply put, “reachable” values are those that are accessible or usable somehow. They are guaranteed to be stored in memory.

There’s a base set of inherently reachable values, that cannot be deleted for obvious reasons.

For instance:

The currently executing function, its local variables and parameters.

Other functions on the current chain of nested calls, their local variables and parameters.

Global variables.

(there are some other, internal ones as well)

These values are called roots.

Any other value is considered reachable if it’s reachable from a root by a reference or by a chain of references.

For instance, if there’s an object in a global variable, and that object has a property referencing another object, that object is considered reachable. And those that it references are also reachable. Detailed examples to follow.

There’s a background process in the JavaScript engine that is called garbage collector. It monitors all objects and removes those that have become unreachable.

A simple example

Here’s the simplest example:

```
    // user has a reference to the object
let user = {
  name: "John"
};
```

Here the arrow depicts an object reference. The global variable "user" references the object {name: "John"} (we'll call it John for brevity). The "name" property of John stores a primitive, so it's painted inside the object. If the value of user is overwritten, the reference is lost:

```
user = null;
```

Now John becomes unreachable. There's no way to access it, no references to it. Garbage collector will junk the data and free the memory.
Two references Now let's imagine we copied the reference from user to admin:

```
    // user has a reference to the object
let user = {
  name: "John"
};

let admin = user;
```

Now if we do the same:

```
user = null;
```

...Then the object is still reachable via admin global variable, so it must stay in memory. If we overwrite admin too, then it can be removed.

Interlinked objectsNow a more complex example. The family:

```
function marry(man, woman) {  
  woman.husband = man;  
  man.wife = woman;  
  
  return {  
    father: man,  
    mother: woman  
  }  
}  
  
let family = marry({  
  name: "John"  
}, {  
  name: "Ann"  
});
```

Function marry “marries” two objects by giving them references to each other and returns a new object that contains them both.

The resulting memory structure:

As of now, all objects are reachable.
Now let's remove two references:

```
delete family.father;  
delete family.mother.husband;
```

It's not enough to delete only one of these two references, because all objects would still be reachable.
But if we delete both, then we can see that John has no incoming reference any more:

Outgoing references do not matter. Only incoming ones can make an object reachable. So, John is now unreachable and will be removed from the memory with all its data that also became inaccessible.
After garbage collection:

Unreachable island
It is possible that the whole island of interlinked objects becomes unreachable and is removed from the memory.
The source object is the same as above. Then:

```
family = null;
```

The in-memory picture becomes:

This example demonstrates how important the concept of reachability is. It's obvious that John and Ann are still linked, both have incoming references. But that's not enough.

The former "family" object has been unlinked from the root, there's no reference to it any more, so the whole island becomes unreachable and will be removed.

Internal algorithmsThe basic garbage collection algorithm is called "mark-and-sweep". The following "garbage collection" steps are regularly performed:

The garbage collector takes roots and "marks" (remembers) them.

Then it visits and "marks" all references from them.

Then it visits marked objects and marks their references. All visited objects are remembered, so as not to visit the same object twice in the future.

...And so on until every reachable (from the roots) references are visited.

All objects except marked ones are removed.

For instance, let our object structure look like this:

We can clearly see an "unreachable island" to the right side. Now let's see how "mark-and-sweep" garbage collector deals with it.

The first step marks the roots:

Then we follow their references and mark referenced objects:

...And continue to follow further references, while possible:

Now the objects that could not be visited in the process are considered unreachable and will be removed:

We can also imagine the process as spilling a huge bucket of paint from the roots, that flows through all references and marks all reachable objects. The unmarked ones are then removed.

That's the concept of how garbage collection works. JavaScript engines apply many optimizations to make it run faster and not introduce any delays into the code execution. Some of the optimizations:

Generational collection – objects are split into two sets: “new ones” and “old ones”. In typical code, many objects have a short life span: they appear, do their job and die fast, so it makes sense to track new objects and clear the memory from them if that's the case. Those that survive for long enough, become “old” and are examined less often.

Incremental collection – if there are many objects, and we try to walk and mark the whole object set at once, it may take some time and introduce visible delays in the execution. So the engine splits the whole set of existing objects into multiple parts. And then clear these parts one after another. There are many small garbage collections instead of a total one. That requires some extra bookkeeping between them to track changes, but we get many tiny delays instead of a big one.

Idle-time collection – the garbage collector tries to run only while the CPU is idle, to reduce the possible effect on the execution.

There exist other optimizations and flavours of garbage collection algorithms. As much as I'd like to describe them here, I have to hold off, because different engines implement different tweaks and techniques. And, what's even more important, things change as engines develop, so studying deeper “in advance”, without a real need is probably not worth that. Unless, of course, it is a matter of pure interest, then there will be some links for you below.

SummaryThe main things to know:

Garbage collection is performed automatically. We cannot force or prevent it.

Objects are retained in memory while they are reachable.

Being referenced is not the same as being reachable (from a root): a pack of interlinked objects can become unreachable as a whole, as we've seen in the example above.

Modern engines implement advanced algorithms of garbage collection.

A general book “The Garbage Collection Handbook: The Art of Automatic Memory Management” (R. Jones et al) covers some of them.

If you are familiar with low-level programming, more detailed information about V8's garbage collector is in the article [A tour of V8: Garbage Collection](#).

The V8 blog also publishes articles about changes in memory management from time

to time. Naturally, to learn more about garbage collection, you'd better prepare by learning about V8 internals in general and read the blog of Vyacheslav Egorov who worked as one of the V8 engineers. I'm saying: "V8", because it is best covered by articles on the internet. For other engines, many approaches are similar, but garbage collection differs in many aspects.

In-depth knowledge of engines is good when you need low-level optimizations. It would be wise to plan that as the next step after you're familiar with the language.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/garbage-collection", "identifier": "Vgarbage-collection" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageObjects: the basics{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Objects: the basics", "item": "https://javascript.info/object-basics"}]}May 22, 2022Object methods, "this"Objects are usually created to represent entities of the real world, like users, orders and so on:

```
let user = {  
  name: "John",  
  age: 30  
};
```

And, in the real world, a user can act: select something from the shopping cart, login, logout etc.

Actions are represented in JavaScript by functions in properties.

Method examplesFor a start, let's teach the user to say hello:

```
    let user = {  
      name: "John",  
      age: 30  
    };  
  
    user.sayHi = function() {  
      alert("Hello!");  
    };  
  
    user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create a function and assign it to the property `user.sayHi` of the object.

Then we can call it as `user.sayHi()`. The user can now speak!

A function that is a property of an object is called its method.

So, here we've got a method `sayHi` of the object `user`.

Of course, we could use a pre-declared function as a method, like this:

```
    let user = {  
      // ...  
    };  
  
    // first, declare  
    function sayHi() {  
      alert("Hello!");  
    }  
  
    // then add as a method
```

```
user.sayHi = sayHi;
```

```
user.sayHi(); // Hello!
```

Object-oriented programming

When we write our code using objects to represent entities, that's called object-oriented programming, in short: "OOP".

OOP is a big thing, an interesting science of its own. How to choose the right entities? How to organize the interaction between them? That's architecture, and there are great books on that topic, like "Design Patterns: Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson, J. Vissides or "Object-Oriented Analysis and Design with Applications" by G. Booch, and more.

Method shorthand There exists a shorter syntax for methods in an object literal:

```
// these objects do the same
```

```
user = {  
  sayHi: function() {  
    alert("Hello");  
  }  
};
```

```
// method shorthand looks better, right?
```

```
user = {  
  sayHi() { // same as "sayHi: function(){...}"  
    alert("Hello");  
  }  
};
```

As demonstrated, we can omit "function" and just write sayHi().

To tell the truth, the notations are not fully identical. There are subtle differences related to object inheritance (to be covered later), but for now they do not matter. In almost all cases, the shorter syntax is preferred.

"this" in methods It's common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside user.sayHi() may need the name of the user.

To access the object, a method can use the this keyword.

The value of this is the object “before dot”, the one used to call the method.
For instance:

```
    let user = {  
    name: "John",  
    age: 30,  
  
    sayHi() {  
      // "this" is the "current object"  
      alert(this.name);  
    }  
  };  
  
user.sayHi(); // John
```

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.
Technically, it's also possible to access the object without `this`, by referencing it via the outer variable:

```
    let user = {  
    name: "John",  
    age: 30,  
  
    sayHi() {  
      alert(user.name); // "user" instead of "this"  
    }  
  };  
  
user.sayHi();
```

...But such code is unreliable. If we decide to copy user to another variable, e.g. admin = user and overwrite user with something else, then it will access the wrong object.

That's demonstrated below:

```
    let user = {
name: "John",
age: 30,

sayHi() {
  alert( user.name ); // leads to an error
}

};
```

```
let admin = user;
user = null; // overwrite to make things obvious
```

```
admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

If we used this.name instead of user.name inside the alert, then the code would work.

“this” is not boundIn JavaScript, keyword this behaves unlike most other programming languages. It can be used in any function, even if it's not a method of an object.

There's no syntax error in the following example:

```
function sayHi() {
```

```
    alert( this.name );  
}
```

The value of `this` is evaluated during the run-time, depending on the context. For instance, here the same function is assigned to two different objects and has different “`this`” in the calls:

```
    let user = { name: "John" };  
    let admin = { name: "Admin" };  
  
    function sayHi() {  
        alert( this.name );  
    }  
  
    // use the same function in two objects  
    user.f = sayHi;  
    admin.f = sayHi;  
  
    // these calls have different this  
    // "this" inside the function is the object "before the dot"  
    user.f(); // John (this == user)  
    admin.f(); // Admin (this == admin)  
  
    admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)
```

The rule is simple: if `obj.f()` is called, then `this` is `obj` during the call of `f`. So it's either `user` or `admin` in the example above.

Calling without an object: `this == undefined`
We can even call the function without an object at all:

```
function sayHi() {  
  alert(this);  
}  
  
sayHi(); // undefined
```

In this case this is undefined in strict mode. If we try to access this.name, there will be an error.

In non-strict mode the value of this in such case will be the global object (window in a browser, we'll get to it later in the chapter Global object). This is a historical behavior that "use strict" fixes.

Usually such call is a programming error. If there's this inside a function, it expects to be called in an object context.

The consequences of unbound this

If you come from another programming language, then you are probably used to the idea of a "bound this", where methods defined in an object always have this referencing that object.

In JavaScript this is "free", its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is "before the dot".

The concept of run-time evaluated this has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, the greater flexibility creates more possibilities for mistakes.

Here our position is not to judge whether this language design decision is good or bad. We'll understand how to work with it, how to get benefits and avoid problems.

Arrow functions have no "this" Arrow functions are special: they don't have their "own" this. If we reference this from such a function, it's taken from the outer "normal" function. For instance, here arrow() uses this from the outer user.sayHi() method:


```

    let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

```

```

user.sayHi(); // Ilya

```

That's a special feature of arrow functions, it's useful when we actually do not want to have a separate this, but rather to take it from the outer context. Later in the chapter Arrow functions revisited we'll go more deeply into arrow functions.

Summary

Functions that are stored in object properties are called "methods".

Methods allow objects to "act" like object.doSomething().

Methods can reference the object as this.

The value of this is defined at run-time.

When a function is declared, it may use this, but that this has no value until the function is called.

A function can be copied between objects.

When a function is called in the "method" syntax: object.method(), the value of this during the call is object.

Please note that arrow functions are special: they have no this. When this is accessed inside an arrow function, it is taken from outside.

TasksUsing "this" in object literalimportance: 5Here the function makeUser returns an object.

What is the result of accessing its ref? Why?

```
function makeUser() {  
  return {  
    name: "John",  
    ref: this  
  };  
}  
  
let user = makeUser();  
  
alert( user.ref.name ); // What's the result?
```

solutionAnswer: an error.
Try it:

```
function makeUser() {  
  return {  
    name: "John",  
    ref: this  
  };  
}  
  
let user = makeUser();  
  
alert( user.ref.name ); // Error: Cannot read property 'name' of undefined
```

That's because rules that set this do not look at object definition. Only the moment of call matters.
Here the value of this inside makeUser() is undefined, because it is called as a function, not as a method with "dot" syntax.
The value of this is one for the whole function, code blocks and object literals do not affect it.

So ref: this actually takes current this of the function.

We can rewrite the function and return the same this with undefined value:

```
function makeUser(){  
  return this; // this time there's no object literal  
}
```

```
alert( makeUser().name ); // Error: Cannot read property 'name' of undefined
```

As you can see the result of `alert(makeUser().name)` is the same as the result of `alert(user.ref.name)` from the previous example.

Here's the opposite case:

```
function makeUser() {  
  return {  
    name: "John",  
    ref() {  
      return this;  
    }  
  };  
}
```

```
let user = makeUser();

alert( user.ref().name ); // John
```

Now it works, because `user.ref()` is a method. And the value of this is set to the object before dot ..

Create a calculator importance: 5 Create an object calculator with three methods:

`read()` prompts for two values and saves them as object properties with names `a` and `b` respectively.

`sum()` returns the sum of saved values.

`mul()` multiplies saved values and returns the result.

```
let calculator = {
  // ... your code ...
};
```

```
calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Run the demo Open a sandbox with tests. solution

```
let calculator = {
  sum() {
    return this.a + this.b;
  },
};
```

```

mul() {
  return this.a * this.b;
},

read() {
  this.a = +prompt('a?', 0);
  this.b = +prompt('b?', 0);
}
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );

```

Open the solution with tests in a sandbox. Chaining importance: 2 There's a ladder object that allows to go up and down:

```

let ladder = {
  step: 0,
  up() {
    this.step++;
  },
  down() {
    this.step--;
  },
  showStep: function() { // shows the current step
    alert( this.step );
  }
};

```

Now, if we need to make several calls in sequence, can do it like this:

```

ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1

```

```
ladder.down();  
ladder.showStep(); // 0
```

Modify the code of up, down and showStep to make the calls chainable, like this:

```
ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

Such approach is widely used across JavaScript libraries.
Open a sandbox with tests.[solution](#)The solution is to return the object itself from every call.

```
let ladder = {  
  step: 0,  
  up() {  
    this.step++;  
    return this;  
  },  
  down() {  
    this.step--;  
    return this;  
  },  
  showStep() {  
    alert( this.step );  
    return this;  
  }  
};
```

```
ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

We also can write a single call per line. For long chains it's more readable:

```
ladder
.up()
.up()
.down()
.showStep() // 1
.down()
.showStep(); // 0
```

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/object-methods","identifier":"Object-methods"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageObjects: the basics{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Objects: the basics","item":"https://javascript.info/object-basics"}]}October 1, 2022Constructor, operator "new"The regular {...} syntax allows us to create one object. But often we need to create many similar objects, like multiple users or menu items and so on. That can be done using constructor functions and the "new" operator. Constructor functionConstructor functions technically are regular functions. There are two conventions though:

They are named with capital letter first.
They should be executed only with "new" operator.

For instance:

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```

When a function is executed with new, it does the following steps:

A new empty object is created and assigned to this.
The function body executes. Usually it modifies this, adds new properties to it.
The value of this is returned.

In other words, new User(...) does something like:

```
function User(name) {  
  // this = {}; (implicitly)  
  
  // add properties to this  
  this.name = name;  
  this.isAdmin = false;  
  
  // return this; (implicitly)  
}
```

So let user = new User("Jack") gives the same result as:


```
    let user = {  
      name: "Jack",  
      isAdmin: false  
    };
```

Now if we want to create other users, we can call `new User("Ann")`, `new User("Alice")` and so on. Much shorter than using literals every time, and also easy to read.

That's the main purpose of constructors – to implement reusable object creation code. Let's note once again – technically, any function (except arrow functions, as they don't have this) can be used as a constructor. It can be run with `new`, and it will execute the algorithm above. The “capital letter first” is a common agreement, to make it clear that a function is to be run with `new`.

```
    new function() { ... }
```

If we have many lines of code all about creation of a single complex object, we can wrap them in an immediately called constructor function, like this:

```
    // create a function and immediately call it with new  
    let user = new function() {  
      this.name = "John";  
      this.isAdmin = false;  
  
      // ...other code for user creation  
      // maybe complex logic and statements  
      // local variables etc  
    };
```

This constructor can't be called again, because it is not saved anywhere, just created and called. So this trick aims to encapsulate the code that constructs the single object, without future reuse.

Constructor mode test: `new.target`

Advanced stuff

The syntax from this section is rarely used, skip it unless you want to know everything.

Inside a function, we can check whether it was called with new or without it, using a special new.target property.

It is undefined for regular calls and equals the function if called with new:

```
function User() {  
  alert(new.target);  
}  
  
// without "new":  
User(); // undefined  
  
// with "new":  
new User(); // function User { ... }
```

That can be used inside the function to know whether it was called with new, “in constructor mode”, or without it, “in regular mode”.

We can also make both new and regular calls to do the same, like this:

```
function User(name) {  
  if (!new.target) { // if you run me without new  
    return new User(name); // ...I will add new for you
```

```

    }

    this.name = name;
}

let john = User("John"); // redirects call to new User
alert(john.name); // John

```

This approach is sometimes used in libraries to make the syntax more flexible. So that people may call the function with or without new, and it still works. Probably not a good thing to use everywhere though, because omitting new makes it a bit less obvious what's going on. With new we all know that the new object is being created.

Return from constructors Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result. But if there is a return statement, then the rule is simple:

If return is called with an object, then the object is returned instead of this.
 If return is called with a primitive, it's ignored.

In other words, return with an object returns that object, in all other cases this is returned.

For instance, here return overrides this by returning an object:

```

function BigUser() {

    this.name = "John";

    return { name: "Godzilla" }; // <-- returns this object
}

alert( new BigUser().name ); // Godzilla, got that object

```

And here's an example with an empty return (or we could place a primitive after it, doesn't matter):

```
function SmallUser() {  
  this.name = "John";  
  return; // <-- returns this  
}  
  
alert( new SmallUser().name ); // John
```

Usually constructors don't have a return statement. Here we mention the special behavior with returning objects mainly for the sake of completeness.

Omitting parentheses

By the way, we can omit parentheses after new:

```
let user = new User; // <-- no parentheses  
// same as  
let user = new User();
```

Omitting parentheses here is not considered a “good style”, but the syntax is permitted by specification.

Methods in constructorUsing constructor functions to create objects gives a great deal

of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to this not only properties, but methods as well.

For instance, `new User(name)` below creates an object with the given name and the method `sayHi`:

```
function User(name) {  
  this.name = name;  
  
  this.sayHi = function() {  
    alert( "My name is: " + this.name );  
  };  
}
```

```
let john = new User("John");
```

```
john.sayHi(); // My name is: John
```

```
/*  
john = {  
  name: "John",  
  sayHi: function() { ... }  
}  
*/
```

To create complex objects, there's a more advanced syntax, classes, that we'll cover later.

Summary

Constructor functions or, briefly, constructors, are regular functions, but there's a common agreement to name them with capital letter first.

Constructor functions should only be called using `new`. Such a call implies a creation of empty this at the start and returning the populated one at the end.

We can use constructor functions to make multiple similar objects. JavaScript provides constructor functions for many built-in language objects: like Date for dates, Set for sets and others that we plan to study.

Objects, we'll be back!

In this chapter we only cover the basics about objects and constructors. They are essential for learning more about data types and functions in the next chapters. After we learn that, we return to objects and cover them in-depth in the chapters Prototypes, inheritance and Classes.

TasksTwo functions – one objectimportance: 2Is it possible to create functions A and B so that `new A() == new B()`?

```
function A() { ... }  
function B() { ... }
```

```
let a = new A();  
let b = new B();
```

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

If it is, then provide an example of their code.

solutionYes, it's possible.

If a function returns an object then new returns it instead of this.

So they can, for instance, return the same externally defined object obj:

```
let obj = {};
```

```
function A() { return obj; }  
function B() { return obj; }
```

```
alert( new A() == new B() ); // true
```

Create new Calculatorimportance: 5Create a constructor function Calculator that creates objects with 3 methods:

read() prompts for two values and saves them as object properties with names a and b respectively.

sum() returns the sum of these properties.

mul() returns the multiplication product of these properties.

For instance:

```
let calculator = new Calculator();  
calculator.read();
```

```
alert( "Sum=" + calculator.sum() );  
alert( "Mul=" + calculator.mul() );
```

Run the demoOpen a sandbox with tests.solution

```
function Calculator() {  
  
  this.read = function() {  
    this.a = +prompt('a?', 0);  
    this.b = +prompt('b?', 0);  
  };  
  
  this.sum = function() {
```

```

    return this.a + this.b;
};

this.mul = function() {
    return this.a * this.b;
};
}

let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );

```

Open the solution with tests in a sandbox. Create new Accumulator importance: 5
 Create a constructor function Accumulator(startingValue).
 Object that it creates should:

Store the “current value” in the property value. The starting value is set to the argument of the constructor startingValue.

The read() method should use prompt to read a new number and add it to value.

In other words, the value property is the sum of all user-entered values with the initial value startingValue.

Here’s the demo of the code:

```

    let accumulator = new Accumulator(1); // initial value 1

    accumulator.read(); // adds the user-entered value
    accumulator.read(); // adds the user-entered value

    alert(accumulator.value); // shows the sum of these values

```

Run the demo Open a sandbox with tests. solution


```

    function Accumulator(startingValue) {
    this.value = startingValue;

    this.read = function() {
    this.value += +prompt('How much to add?', 0);
    };

    }

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);

```

Open the solution with tests in a sandbox. Ctrl + ! • Previous lesson Ctrl + ! • Next lesson

Share Tutorial map Comments read this before commenting... If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url": "https://javascript.info/vconstructor-new", "identifier": "vconstructor-new"}); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true; Tutorial The JavaScript language Objects: the basics {"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Objects: the basics", "item": "https://javascript.info/object-basics"}]} April 14, 2022 Optional chaining '?.'

A recent addition

This is a recent addition to the language.
Old browsers may need polyfills.

The optional chaining `?.` is a safe way to access nested object properties, even if an intermediate property doesn't exist.

The “non-existing property” problem If you've just started to read the tutorial and learn JavaScript, maybe the problem hasn't touched you yet, but it's quite common.

As an example, let's say we have user objects that hold the information about our users. Most of our users have addresses in user.address property, with the street user.address.street, but some did not provide them. In such case, when we attempt to get user.address.street, and the user happens to be without an address, we get an error:

```
let user = {}; // a user without "address" property  
  
alert(user.address.street); // Error!
```

That's the expected result. JavaScript works like this. As user.address is undefined, an attempt to get user.address.street fails with an error. In many practical cases we'd prefer to get undefined instead of an error here (meaning "no street").

...and another example. In Web development, we can get an object that corresponds to a web page element using a special method call, such as document.querySelector('.elem'), and it returns null when there's no such element.

```
// document.querySelector('.elem') is null if there's no element  
let html = document.querySelector('.elem').innerHTML; // error if it's null
```

Once again, if the element doesn't exist, we'll get an error accessing `.innerHTML` property of `null`. And in some cases, when the absence of the element is normal, we'd like to avoid the error and just accept `html = null` as the result.

How can we do this?

The obvious solution would be to check the value using `if` or the conditional operator `?`, before accessing its property, like this:

```
let user = {};  
  
alert(user.address ? user.address.street : undefined);
```

It works, there's no error... But it's quite inelegant. As you can see, the `"user.address"` appears twice in the code.
Here's how the same would look for `document.querySelector`:

```
let html = document.querySelector('.elem') ?  
document.querySelector('.elem').innerHTML : null;
```

We can see that the element search `document.querySelector('.elem')` is actually called twice here. Not good.

For more deeply nested properties, it becomes even uglier, as more repetitions are required.

E.g. let's get `user.address.street.name` in a similar fashion.

```
let user = {}; // user has no address  
  
alert(user.address ? user.address.street ? user.address.street.name : null : null);
```

That's just awful, one may even have problems understanding such code.
There's a little better way to write it, using the && operator:

```
let user = {}; // user has no address  
  
alert( user.address && user.address.street && user.address.street.name ); // undefined  
(no error)
```

AND'ing the whole path to the property ensures that all components exist (if not, the evaluation stops), but also isn't ideal.

As you can see, property names are still duplicated in the code. E.g. in the code above, `user.address` appears three times.

That's why the optional chaining `?.` was added to the language. To solve this problem once and for all!

Optional chaining The optional chaining `?.` stops the evaluation if the value before `?.` is undefined or null and returns undefined.

Further in this article, for brevity, we'll be saying that something "exists" if it's not null and not undefined.

In other words, `value?.prop`:

works as `value.prop`, if value exists,
otherwise (when value is undefined/null) it returns undefined.

Here's the safe way to access `user.address.street` using `?.`:

```
let user = {}; // user has no address  
alert( user?.address?.street ); // undefined (no error)
```

The code is short and clean, there's no duplication at all.
Here's an example with `document.querySelector`:

```
let html = document.querySelector('.elem')?.innerHTML; // will be undefined, if  
there's no element
```

Reading the address with `user?.address` works even if user object doesn't exist:

```
let user = null;

alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

Please note: the `?.` syntax makes optional the value before it, but not any further. E.g. in `user?.address.street.name` the `?.` allows `user` to safely be `null/undefined` (and returns `undefined` in that case), but that's only for `user`. Further properties are accessed in a regular way. If we want some of them to be optional, then we'll need to replace more `.` with `?.`.

Don't overuse the optional chaining

We should use `?.` only where it's ok that something doesn't exist.

For example, if according to our code logic `user` object must exist, but `address` is optional, then we should write `user.address?.street`, but not `user?.address?.street`. Then, if `user` happens to be `undefined`, we'll see a programming error about it and fix it. Otherwise, if we overuse `?.`, coding errors can be silenced where not appropriate, and become more difficult to debug.

The variable before `?.` must be declared

If there's no variable `user` at all, then `user?.anything` triggers an error:

```
// ReferenceError: user is not defined
user?.address;
```

The variable must be declared (e.g. `let/const/var user` or as a function parameter). The optional chaining works only for declared variables.

Short-circuiting As it was said before, the `?.` immediately stops ("short-circuits") the

evaluation if the left part doesn't exist.

So, if there are any further function calls or operations to the right of `?.`, they won't be made.

For instance:

```
let user = null;
let x = 0;

user?.sayHi(x++); // no "user", so the execution doesn't reach sayHi call and x++

alert(x); // 0, value not incremented
```

Other variants: `?.()`, `?.[]` The optional chaining `?.` is not an operator, but a special syntax construct, that also works with functions and square brackets.

For example, `?.()` is used to call a function that may not exist.

In the code below, some of our users have admin method, and some don't:

```
let userAdmin = {
  admin() {
    alert("I am admin");
  }
};
```

```
let userGuest = {};  
  
userAdmin.admin?.(); // I am admin  
  
userGuest.admin?.(); // nothing happens (no such method)
```

Here, in both lines we first use the dot (`userAdmin.admin`) to get `admin` property, because we assume that the user object exists, so it's safe read from it. Then `?.()` checks the left part: if the `admin` function exists, then it runs (that's so for `userAdmin`). Otherwise (for `userGuest`) the evaluation stops without errors. The `?.[]` syntax also works, if we'd like to use brackets `[]` to access properties instead of dot `..`. Similar to previous cases, it allows to safely read a property from an object that may not exist.

```
let key = "firstName";  
  
let user1 = {  
  firstName: "John"  
};  
  
let user2 = null;  
  
alert( user1?.[key] ); // John  
alert( user2?.[key] ); // undefined
```

Also we can use `?.` with `delete`:


```
delete user?.name; // delete user.name if user exists
```

We can use `?.` for safe reading and deleting, but not writing
The optional chaining `?.` has no use on the left side of an assignment.
For example:

```
let user = null;  
  
user?.name = "John"; // Error, doesn't work  
// because it evaluates to: undefined = "John"
```

SummaryThe optional chaining `?.` syntax has three forms:

`obj?.prop` – returns `obj.prop` if `obj` exists, otherwise `undefined`.
`obj?.[prop]` – returns `obj[prop]` if `obj` exists, otherwise `undefined`.
`obj.method?.()` – calls `obj.method()` if `obj.method` exists, otherwise returns `undefined`.

As we can see, all of them are straightforward and simple to use. The `?.` checks the left part for `null/undefined` and allows the evaluation to proceed if it's not so.

A chain of `?.` allows to safely access nested properties.

Still, we should apply `?.` carefully, only where it's acceptable, according to our code

logic, that the left part doesn't exist. So that it won't hide programming errors from us, if they occur.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/optional-chaining", "identifier": "optional-chaining" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageObjects: the basics{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Objects: the basics", "item": "https://javascript.info/object-basics"}]}June 27, 2022Symbol typeBy specification, only two primitive types may serve as object property keys:

string type, or
symbol type.

Otherwise, if one uses another type, such as number, it's autoconverted to string. So that obj[1] is the same as obj["1"], and obj[true] is the same as obj["true"].

Until now we've been using only strings.

Now let's explore symbols, see what they can do for us.

SymbolsA "symbol" represents a unique identifier.

A value of this type can be created using Symbol():

```
let id = Symbol();
```

Upon creation, we can give symbols a description (also called a symbol name), mostly useful for debugging purposes:

```
// id is a symbol with the description "id"  
let id = Symbol("id");
```

Symbols are guaranteed to be unique. Even if we create many symbols with exactly the same description, they are different values. The description is just a label that doesn't affect anything.

For instance, here are two symbols with the same description – they are not equal:

```
let id1 = Symbol("id");  
let id2 = Symbol("id");  
  
alert(id1 == id2); // false
```

If you are familiar with Ruby or another language that also has some sort of “symbols” – please don't be misguided. JavaScript symbols are different. So, to summarize, a symbol is a “primitive unique value” with an optional description. Let's see where we can use them.

Symbols don't auto-convert to a string

Most values in JavaScript support implicit conversion to a string. For instance, we can alert almost any value, and it will work. Symbols are special. They don't auto-convert.

For instance, this alert will show an error:

```
let id = Symbol("id");  
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

That's a "language guard" against messing up, because strings and symbols are fundamentally different and should not accidentally convert one into another. If we really want to show a symbol, we need to explicitly call `.toString()` on it, like here:

```
let id = Symbol("id");  
alert(id.toString()); // Symbol(id), now it works
```

Or get `symbol.description` property to show the description only:

```
let id = Symbol("id");  
alert(id.description); // id
```

"Hidden" properties Symbols allow us to create "hidden" properties of an object, that no other part of code can accidentally access or overwrite.

For instance, if we're working with user objects, that belong to a third-party code. We'd like to add identifiers to them.
Let's use a symbol key for it:

```
    let user = { // belongs to another code
name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // we can access the data using the symbol as the key
```

What's the benefit of using `Symbol("id")` over a string `"id"`?
As user objects belong to another codebase, it's unsafe to add fields to them, since we might affect pre-defined behavior in that other codebase. However, symbols cannot be accessed accidentally. The third-party code won't be aware of newly defined symbols, so it's safe to add symbols to the user objects.
Also, imagine that another script wants to have its own identifier inside user, for its own purposes.
Then that script can create its own `Symbol("id")`, like this:

```
    // ...
let id = Symbol("id");

user[id] = "Their id value";
```

There will be no conflict between our and their identifiers, because symbols are always different, even if they have the same name.
...But if we used a string "id" instead of a symbol for the same purpose, then there would be a conflict:

```
let user = { name: "John" };

// Our script uses "id" property
user.id = "Our id value";

// ...Another script also wants "id" for its purposes...

user.id = "Their id value"
// Boom! overwritten by another script!
```

Symbols in an object literalIf we want to use a symbol in an object literal {...}, we need square brackets around it.
Like this:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not "id": 123
};
```

That's because we need the value from the variable id as the key, not the string "id". Symbols are skipped by for...inSymbolic properties do not participate in for..in loop.
For instance:

```
    let id = Symbol("id");  
let user = {  
  name: "John",  
  age: 30,  
  [id]: 123  
};
```

```
for (let key in user) alert(key); // name, age (no symbols)
```

```
// the direct access by the symbol works  
alert( "Direct: " + user[id] ); // Direct: 123
```

Object.keys(user) also ignores them. That's a part of the general “hiding symbolic properties” principle. If another script or a library loops over our object, it won't unexpectedly access a symbolic property.

In contrast, Object.assign copies both string and symbol properties:

```
    let id = Symbol("id");  
let user = {  
  [id]: 123  
};
```

```
let clone = Object.assign({}, user);
```

```
alert( clone[id] ); // 123
```

There's no paradox here. That's by design. The idea is that when we clone an object or merge objects, we usually want all properties to be copied (including symbols like `id`). Global symbolsAs we've seen, usually all symbols are different, even if they have the same name. But sometimes we want same-named symbols to be same entities. For instance, different parts of our application want to access symbol `"id"` meaning exactly the same property.

To achieve that, there exists a global symbol registry. We can create symbols in it and access them later, and it guarantees that repeated accesses by the same name return exactly the same symbol.

In order to read (create if absent) a symbol from the registry, use `Symbol.for(key)`.

That call checks the global registry, and if there's a symbol described as `key`, then returns it, otherwise creates a new symbol `Symbol(key)` and stores it in the registry by the given key.

For instance:

```
// read from the global registry
let id = Symbol.for("id"); // if the symbol did not exist, it is created

// read it again (maybe from another part of the code)
let idAgain = Symbol.for("id");

// the same symbol
alert( id === idAgain ); // true
```

Symbols inside the registry are called global symbols. If we want an application-wide symbol, accessible everywhere in the code – that's what they are for.

That sounds like Ruby

In some programming languages, like Ruby, there's a single symbol per name. In JavaScript, as we can see, that's true for global symbols.

`Symbol.keyFor`We have seen that for global symbols, `Symbol.for(key)` returns a symbol by name. To do the opposite – return a name by global symbol – we can use:

`Symbol.keyFor(sym)`:

For instance:

```
// get symbol by name
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// get name by symbol
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

The `Symbol.keyFor` internally uses the global symbol registry to look up the key for the symbol. So it doesn't work for non-global symbols. If the symbol is not global, it won't be able to find it and returns undefined.

That said, all symbols have the `description` property.

For instance:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, global symbol
alert( Symbol.keyFor(localSymbol) ); // undefined, not global
```

```
alert( localSymbol.description ); // name
```

System symbols There exist many “system” symbols that JavaScript uses internally, and we can use them to fine-tune various aspects of our objects. They are listed in the specification in the Well-known symbols table:

Symbol.hasInstance
Symbol.isConcatSpreadable
Symbol.iterator
Symbol.toPrimitive
...and so on.

For instance, Symbol.toPrimitive allows us to describe object to primitive conversion. We’ll see its use very soon.

Other symbols will also become familiar when we study the corresponding language features.

Summary Symbol is a primitive type for unique identifiers.

Symbols are created with Symbol() call with an optional description (name).

Symbols are always different values, even if they have the same name. If we want same-named symbols to be equal, then we should use the global registry:

Symbol.for(key) returns (creates if needed) a global symbol with key as the name.

Multiple calls of Symbol.for with the same key return exactly the same symbol.

Symbols have two main use cases:

“Hidden” object properties.

If we want to add a property into an object that “belongs” to another script or a library, we can create a symbol and use it as a property key. A symbolic property does not appear in for..in, so it won’t be accidentally processed together with other properties. Also it won’t be accessed directly, because another script does not have our symbol. So the property will be protected from accidental use or overwrite.

So we can “covertly” hide something into objects that we need, but others should not see, using symbolic properties.

There are many system symbols used by JavaScript which are accessible as Symbol.*. We can use them to alter some built-in behaviors. For instance, later in the tutorial we’ll use Symbol.iterator for iterables, Symbol.toPrimitive to setup object-to-primitive conversion and so on.

Technically, symbols are not 100% hidden. There is a built-in method Object.getOwnPropertySymbols(obj) that allows us to get all symbols. Also there is a method named Reflect.ownKeys(obj) that returns all keys of an object including

symbolic ones. But most libraries, built-in functions and syntax constructs don't use these methods.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/symbol","identifier":"Vsymbol"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialThe JavaScript languageObjects: the basics{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}," {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}," {"@type":"ListItem","position":3,"name":"Objects: the basics","item":"https://javascript.info/object-basics"}]}October 5, 2022Object to primitive conversionWhat happens when objects are added obj1 + obj2, subtracted obj1 - obj2 or printed using alert(obj)?

JavaScript doesn't allow you to customize how operators work on objects. Unlike some other programming languages, such as Ruby or C++, we can't implement a special object method to handle addition (or other operators).

In case of such operations, objects are auto-converted to primitives, and then the operation is carried out over these primitives and results in a primitive value.

That's an important limitation: the result of obj1 + obj2 (or another math operation) can't be another object!

E.g. we can't make objects representing vectors or matrices (or achievements or whatever), add them and expect a "summed" object as the result. Such architectural feats are automatically "off the board".

So, because we can't technically do much here, there's no maths with objects in real projects. When it happens, with rare exceptions, it's because of a coding mistake.

In this chapter we'll cover how an object converts to primitive and how to customize it. We have two purposes:

It will allow us to understand what's going on in case of coding mistakes, when such an operation happened accidentally.

There are exceptions, where such operations are possible and look good. E.g. subtracting or comparing dates (Date objects). We'll come across them later.

Conversion rulesIn the chapter Type Conversions we've seen the rules for numeric, string and boolean conversions of primitives. But we left a gap for objects. Now, as we know about methods and symbols it becomes possible to fill it.

There's no conversion to boolean. All objects are true in a boolean context, as simple as that. There exist only numeric and string conversions.

The numeric conversion happens when we subtract objects or apply mathematical functions. For instance, Date objects (to be covered in the chapter Date and time) can

be subtracted, and the result of `date1 - date2` is the time difference between two dates. As for the string conversion – it usually happens when we output an object with `alert(obj)` and in similar contexts.

We can implement string and numeric conversion by ourselves, using special object methods.

Now let's get into technical details, because it's the only way to cover the topic in-depth.

HintsHow does JavaScript decide which conversion to apply?

There are three variants of type conversion, that happen in various situations. They're called "hints", as described in the specification:

"string"

For an object-to-string conversion, when we're doing an operation on an object that expects a string, like `alert`:

```
// output
alert(obj);
```

```
// using object as a property key
anotherObj[obj] = 123;
```

"number"

For an object-to-number conversion, like when we're doing maths:

```
// explicit conversion
let num = Number(obj);
```

```
// maths (except binary plus)
let n = +obj; // unary plus
let delta = date1 - date2;
```

```
// less/greater comparison
let greater = user1 > user2;
```

Most built-in mathematical functions also include such conversion.

"default"

Occurs in rare cases when the operator is “not sure” what type to expect. For instance, binary plus + can work both with strings (concatenates them) and numbers (adds them). So if a binary plus gets an object as an argument, it uses the "default" hint to convert it.

Also, if an object is compared using == with a string, number or a symbol, it's also unclear which conversion should be done, so the "default" hint is used.

```
// binary plus uses the "default" hint
let total = obj1 + obj2;
```

```
// obj == number uses the "default" hint
if (user == 1) { ... };
```

The greater and less comparison operators, such as < >, can work with both strings and numbers too. Still, they use the "number" hint, not "default". That's for historical reasons.

In practice though, things are a bit simpler.

All built-in objects except for one case (Date object, we'll learn it later) implement "default" conversion the same way as "number". And we probably should do the same. Still, it's important to know about all 3 hints, soon we'll see why.

To do the conversion, JavaScript tries to find and call three object methods:

Call obj[Symbol.toPrimitive](hint) – the method with the symbolic key Symbol.toPrimitive (system symbol), if such method exists,
Otherwise if hint is "string"

try calling obj.toString() or obj.valueOf(), whatever exists.

Otherwise if hint is "number" or "default"

try calling obj.valueOf() or obj.toString(), whatever exists.

Symbol.toPrimitiveLet's start from the first method. There's a built-in symbol named Symbol.toPrimitive that should be used to name the conversion method, like this:

```
obj[Symbol.toPrimitive] = function(hint) {  
  // here goes the code to convert this object to a primitive  
  // it must return a primitive value  
  // hint = one of "string", "number", "default"  
};
```

If the method Symbol.toPrimitive exists, it's used for all hints, and no more methods are needed.

For instance, here user object implements it:

```
let user = {  
  name: "John",  
  money: 1000,  
  
  [Symbol.toPrimitive](hint) {  
    alert(`hint: ${hint}`);  
    return hint == "string" ? `{name: "${this.name}"}` : this.money;  
  }  
};
```

```
// conversions demo:  
alert(user); // hint: string -> {name: "John"}  
alert(+user); // hint: number -> 1000  
alert(user + 500); // hint: default -> 1500
```

As we can see from the code, user becomes a self-descriptive string or a money amount, depending on the conversion. The single method `user[Symbol.toPrimitive]` handles all conversion cases.

`toString/valueOf` If there's no `Symbol.toPrimitive` then JavaScript tries to find methods `toString` and `valueOf`:

For the "string" hint: call `toString` method, and if it doesn't exist or if it returns an object instead of a primitive value, then call `valueOf` (so `toString` has the priority for string conversions).

For other hints: call `valueOf`, and if it doesn't exist or if it returns an object instead of a primitive value, then call `toString` (so `valueOf` has the priority for maths).

Methods `toString` and `valueOf` come from ancient times. They are not symbols (symbols did not exist that long ago), but rather "regular" string-named methods. They provide an alternative "old-style" way to implement the conversion.

These methods must return a primitive value. If `toString` or `valueOf` returns an object, then it's ignored (same as if there were no method).

By default, a plain object has following `toString` and `valueOf` methods:

The `toString` method returns a string `"[object Object]"`.

The `valueOf` method returns the object itself.

Here's the demo:

```
let user = {name: "John"};

alert(user); // [object Object]
alert(user.valueOf() === user); // true
```

So if we try to use an object as a string, like in an alert or so, then by default we see `[object Object]`.

The default `valueOf` is mentioned here only for the sake of completeness, to avoid any

confusion. As you can see, it returns the object itself, and so is ignored. Don't ask me why, that's for historical reasons. So we can assume it doesn't exist.

Let's implement these methods to customize the conversion.

For instance, here user does the same as above using a combination of toString and valueOf instead of Symbol.toPrimitive:

```
    let user = {
  name: "John",
  money: 1000,

  // for hint="string"
  toString() {
    return `${name: "${this.name}"}`;
  },

  // for hint="number" or "default"
  valueOf() {
    return this.money;
  }
};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

As we can see, the behavior is the same as the previous example with Symbol.toPrimitive.

Often we want a single “catch-all” place to handle all primitive conversions. In this case, we can implement toString only, like this:


```
    let user = {  
      name: "John",  
  
      toString() {  
        return this.name;  
      }  
    };  
  
    alert(user); // toString -> John  
    alert(user + 500); // toString -> John500
```

In the absence of `Symbol.toPrimitive` and `valueOf`, `toString` will handle all primitive conversions.

A conversion can return any primitive type. The important thing to know about all primitive-conversion methods is that they do not necessarily return the “hinted” primitive. There is no control whether `toString` returns exactly a string, or whether `Symbol.toPrimitive` method returns a number for the hint "number". The only mandatory thing: these methods must return a primitive, not an object.

Historical notes

For historical reasons, if `toString` or `valueOf` returns an object, there’s no error, but such value is ignored (like if the method didn’t exist). That’s because in ancient times there was no good “error” concept in JavaScript.

In contrast, `Symbol.toPrimitive` is stricter, it must return a primitive, otherwise there will be an error.

Further conversions As we know already, many operators and functions perform type conversions, e.g. multiplication `*` converts operands to numbers.

If we pass an object as an argument, then there are two stages of calculations:

The object is converted to a primitive (using the rules described above).

If necessary for further calculations, the resulting primitive is also converted.

For instance:

```
    let obj = {  
    // toString handles all conversions in the absence of other methods  
    toString() {  
        return "2";  
    }  
};
```

`alert(obj * 2);` // 4, object converted to primitive "2", then multiplication made it a number

The multiplication `obj * 2` first converts the object to primitive (that's a string "2"). Then `"2" * 2` becomes `2 * 2` (the string is converted to number).

Binary plus will concatenate strings in the same situation, as it gladly accepts a string:

```
    let obj = {  
    toString() {  
        return "2";  
    }  
};
```

`alert(obj + 2);` // 22 ("2" + 2), conversion to primitive returned a string => concatenation

SummaryThe object-to-primitive conversion is called automatically by many built-in functions and operators that expect a primitive as a value.
There are 3 types (hints) of it:

"string" (for alert and other operations that need a string)

"number" (for maths)

"default" (few operators, usually objects implement it the same way as "number")

The specification describes explicitly which operator uses which hint.
The conversion algorithm is:

Call obj[Symbol.toPrimitive](hint) if the method exists,
Otherwise if hint is "string"

try calling obj.toString() or obj.valueOf(), whatever exists.

Otherwise if hint is "number" or "default"

try calling obj.valueOf() or obj.toString(), whatever exists.

All these methods must return a primitive to work (if defined).

In practice, it's often enough to implement only obj.toString() as a “catch-all” method for string conversions that should return a “human-readable” representation of an object, for logging or debugging purposes.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/object-to-primitive", "identifier": "object-to-primitive" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript language{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}]}Data typesMore data structures and more in-depth study of the types.

Methods of primitivesNumbersStringsArraysArray methodsIterablesMap and

SetWeakMap and WeakSetObject.keys, values, entriesDestructuring assignmentDate and timeJSON methods, toJSONCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map

TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":

```
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Data types","item":"https://javascript.info/data-types"]}]
```

June 12, 2022Methods of primitivesJavaScript allows us to work with primitives (strings, numbers, etc.) as if they were objects. They also provide methods to call as such. We will study those soon, but first we'll see how it works because, of course, primitives are not objects (and here we will make it even clearer). Let's look at the key distinctions between primitives and objects.

A primitive

Is a value of a primitive type.

There are 7 primitive types: string, number, bigint, boolean, symbol, null and undefined.

An object

Is capable of storing multiple values as properties.

Can be created with {}, for instance: {name: "John", age: 30}. There are other kinds of objects in JavaScript: functions, for example, are objects.

One of the best things about objects is that we can store a function as one of its properties.

```
let john = {
  name: "John",
  sayHi: function() {
    alert("Hi buddy!");
  }
};

john.sayHi(); // Hi buddy!
```

So here we've made an object john with the method sayHi.
Many built-in objects already exist, such as those that work with dates, errors, HTML elements, etc. They have different properties and methods.
But, these features come with a cost!
Objects are "heavier" than primitives. They require additional resources to support the internal machinery.
A primitive as an object Here's the paradox faced by the creator of JavaScript:

There are many things one would want to do with a primitive, like a string or a number.
It would be great to access them using methods.
Primitives must be as fast and lightweight as possible.

The solution looks a little bit awkward, but here it is:

Primitives are still primitive. A single value, as desired.
The language allows access to methods and properties of strings, numbers, booleans and symbols.
In order for that to work, a special "object wrapper" that provides the extra functionality is created, and then is destroyed.

The "object wrappers" are different for each primitive type and are called: String, Number, Boolean, Symbol and BigInt. Thus, they provide different sets of methods.
For instance, there exists a string method str.toUpperCase() that returns a capitalized str.
Here's how it works:

```
let str = "Hello";  
  
alert( str.toUpperCase() ); // HELLO
```

Simple, right? Here's what actually happens in `str.toUpperCase()`:

The string `str` is a primitive. So in the moment of accessing its property, a special object is created that knows the value of the string, and has useful methods, like `toUpperCase()`.

That method runs and returns a new string (shown by `alert()`).

The special object is destroyed, leaving the primitive `str` alone.

So primitives can provide methods, but they still remain lightweight.

The JavaScript engine highly optimizes this process. It may even skip the creation of the extra object at all. But it must still adhere to the specification and behave as if it creates one.

A number has methods of its own, for instance, `toFixed(n)` rounds the number to the given precision:

```
let n = 1.23456;  
  
alert( n.toFixed(2) ); // 1.23
```

We'll see more specific methods in chapters Numbers and Strings.

Constructors `String/Number/Boolean` are for internal use only

Some languages like Java allow us to explicitly create "wrapper objects" for primitives using a syntax like `new Number(1)` or `new Boolean(false)`.

In JavaScript, that's also possible for historical reasons, but highly unrecommended.

Things will go crazy in several places.

For instance:

```
alert( typeof 0 ); // "number"
alert( typeof new Number(0) ); // "object"!
```

Objects are always truthy in if, so here the alert will show up:

```
let zero = new Number(0);
if (zero) { // zero is true, because it's an object
  alert( "zero is truthy!?!" );
}
```

On the other hand, using the same functions String/Number/Boolean without new is totally fine and useful thing. They convert a value to the corresponding type: to a string, a number, or a boolean (primitive). For example, this is entirely valid:

```
let num = Number("123"); // convert a string to number
```

null/undefined have no methods

The special primitives null and undefined are exceptions. They have no corresponding “wrapper objects” and provide no methods. In a sense, they are “the most primitive”.

An attempt to access a property of such value would give the error:

```
alert(null.test); // error
```

Summary

Primitives except null and undefined provide many helpful methods. We will study those in the upcoming chapters.

Formally, these methods work via temporary objects, but JavaScript engines are well tuned to optimize that internally, so they are not expensive to call.

TasksCan I add a string property?importance: 5Consider the following code:

```
let str = "Hello";  
  
str.test = 5;  
  
alert(str.test);
```

What do you think, will it work? What will be shown?
solutionTry running it:


```

let str = "Hello";

str.test = 5; // (*)

alert(str.test);

```

Depending on whether you have use strict or not, the result may be:

undefined (no strict mode)
 An error (strict mode).

Why? Let's replay what's happening at line (*):

When a property of str is accessed, a “wrapper object” is created.
 In strict mode, writing into it is an error.
 Otherwise, the operation with the property is carried on, the object gets the test property, but after that the “wrapper object” disappears, so in the last line str has no trace of the property.

This example clearly shows that primitives are not objects.
 They can't store additional data.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};
 Object.assign(this.page, {"url":"https://\V\javascript.info\primitives-methods","identifier":"\V\primitives-methods"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;
 TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
 [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
 {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://

```
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Data types", "item": "https://  
javascript.info/data-types"}]]}October 1, 2022Numbers
```

In modern JavaScript, there are two types of numbers:

Regular numbers in JavaScript are stored in 64-bit format IEEE-754, also known as “double precision floating point numbers”. These are numbers that we’re using most of the time, and we’ll talk about them in this chapter.

BigInt numbers represent integers of arbitrary length. They are sometimes needed because a regular integer number can’t safely exceed $(2^{53}-1)$ or be less than $-(2^{53}-1)$, as we mentioned earlier in the chapter Data types. As bigints are used in few special areas, we devote them a special chapter BigInt.

So here we’ll talk about regular numbers. Let’s expand our knowledge of them.
More ways to write a numberImagine we need to write 1 billion. The obvious way is:

```
let billion = 1000000000;
```

We also can use underscore `_` as the separator:

```
let billion = 1_000_000_000;
```

Here the underscore `_` plays the role of the “syntactic sugar”, it makes the number more readable. The JavaScript engine simply ignores `_` between digits, so it’s exactly the same one billion as above.

In real life though, we try to avoid writing long sequences of zeroes. We’re too lazy for that. We’ll try to write something like “1bn” for a billion or “7.3bn” for 7 billion 300 million. The same is true for most large numbers.

In JavaScript, we can shorten a number by appending the letter “e” to it and specifying the zeroes count:

```
let billion = 1e9; // 1 billion, literally: 1 and 9 zeroes  
alert( 7.3e9 ); // 7.3 billions (same as 7300000000 or 7_300_000_000)
```

In other words, e multiplies the number by 1 with the given zeroes count.

```
1e3 === 1 * 1000; // e3 means *1000  
1.23e6 === 1.23 * 1000000; // e6 means *1000000
```

Now let's write something very small. Say, 1 microsecond (one millionth of a second):

```
let mD 2 Ò ã °
```

Just like before, using "e" can help. If we'd like to avoid writing the zeroes explicitly, we could write the same as:

```
let mcs = 1e-6; // five zeroes to the left from 1
```

If we count the zeroes in 0.000001, there are 6 of them. So naturally it's 1e-6.

In other words, a negative number after "e" means a division by 1 with the given number of zeroes:

```
// -3 divides by 1 with 3 zeroes  
1e-3 === 1 / 1000; // 0.001
```

```
// -6 divides by 1 with 6 zeroes  
1.23e-6 === 1.23 / 1000000; // 0.00000123
```

```
// an example with a bigger number  
1234e-2 === 1234 / 100; // 12.34, decimal point moves 2 times
```

Hex, binary and octal numbersHexadecimal numbers are widely used in JavaScript to represent colors, encode characters, and for many other things. So naturally, there exists a shorter way to write them: 0x and then the number. For instance:

```
alert( 0xff ); // 255  
alert( 0xFF ); // 255 (the same, case doesn't matter)
```

Binary and octal numeral systems are rarely used, but also supported using the 0b and 0o prefixes:

```
let a = 0b11111111; // binary form of 255
let b = 0o377; // octal form of 255

alert( a == b ); // true, the same number 255 at both sides
```

There are only 3 numeral systems with such support. For other numeral systems, we should use the function `parseInt` (which we will see later in this chapter).

`toString(base)` The method `num.toString(base)` returns a string representation of `num` in the numeral system with the given base.

For example:

```
let num = 255;

alert( num.toString(16) ); // ff
alert( num.toString(2) ); // 11111111
```

The base can vary from 2 to 36. By default it's 10. Common use cases for this are:

`base=16` is used for hex colors, character encodings etc, digits can be 0..9 or A..F.

`base=2` is mostly for debugging bitwise operations, digits can be 0 or 1.

base=36 is the maximum, digits can be 0..9 or A..Z. The whole latin alphabet is used to represent a number. A funny, but useful case for 36 is when we need to turn a long numeric identifier into something shorter, for example to make a short url. Can simply represent it in the numeral system with base 36:

```
alert( 123456..toString(36) ); // 2n9c
```

Two dots to call a method

Please note that two dots in 123456..toString(36) is not a typo. If we want to call a method directly on a number, like toString in the example above, then we need to place two dots .. after it.

If we placed a single dot: 123456.toString(36), then there would be an error, because JavaScript syntax implies the decimal part after the first dot. And if we place one more dot, then JavaScript knows that the decimal part is empty and now goes the method. Also could write (123456).toString(36).

RoundingOne of the most used operations when working with numbers is rounding. There are several built-in functions for rounding:

Math.floor

Rounds down: 3.1 becomes 3, and -1.1 becomes -2.

Math.ceil

Rounds up: 3.1 becomes 4, and -1.1 becomes -1.

Math.round

Rounds to the nearest integer: 3.1 becomes 3, 3.6 becomes 4, the middle case: 3.5 rounds up to 4 too.

Math.trunc (not supported by Internet Explorer)

Removes anything after the decimal point without rounding: 3.1 becomes 3, -1.1 becomes -1.

Here's the table to summarize the differences between them:

Math.floor
Math.ceil
Math.round
Math.trunc

3.1
3
4
3
3

3.6
3
4
4
3

-1.1
-2
-1
-1
-1

-1.6
-2
-1
-2
-1

These functions cover all of the possible ways to deal with the decimal part of a number. But what if we'd like to round the number to n-th digit after the decimal? For instance, we have 1.2345 and want to round it to 2 digits, getting only 1.23.

There are two ways to do so:

Multiply-and-divide.

For example, to round the number to the 2nd digit after the decimal, we can multiply the number by 100, call the rounding function and then divide it back.

```
let num = 1.23456;  
  
alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

The method `toFixed(n)` rounds the number to `n` digits after the point and returns a string representation of the result.

```
let num = 12.34;  
alert( num.toFixed(1) ); // "12.3"
```

This rounds up or down to the nearest value, similar to `Math.round`:


```
let num = 12.36;  
alert( num.toFixed(1) ); // "12.4"
```

Please note that the result of `toFixed` is a string. If the decimal part is shorter than required, zeroes are appended to the end:

```
let num = 12.34;  
alert( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digits
```

We can convert it to a number using the unary plus or a `Number()` call, e.g write `+num.toFixed(5)`.

Imprecise calculations Internally, a number is represented in 64-bit format IEEE-754, so there are exactly 64 bits to store a number: 52 of them are used to store the digits, 11 of them store the position of the decimal point, and 1 bit is for the sign. If a number is really huge, it may overflow the 64-bit storage and become a special numeric value `Infinity`:

```
alert( 1e500 ); // Infinity
```

What may be a little less obvious, but happens quite often, is the loss of precision. Consider this (falsy!) equality test:

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

That's right, if we check whether the sum of 0.1 and 0.2 is 0.3, we get false. Strange! What is it then if not 0.3?

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

Ouch! Imagine you're making an e-shopping site and the visitor puts \$0.10 and \$0.20 goods into their cart. The order total will be \$0.30000000000000004. That would surprise anyone.

But why does this happen?

A number is stored in memory in its binary form, a sequence of bits – ones and zeroes. But fractions like 0.1, 0.2 that look simple in the decimal numeric system are actually unending fractions in their binary form.

What is 0.1? It is one divided by ten $1/10$, one-tenth. In decimal numeral system such numbers are easily representable. Compare it to one-third: $1/3$. It becomes an endless fraction $0.33333(3)$.

So, division by powers of 10 is guaranteed to work well in the decimal system, but division by 3 is not. For the same reason, in the binary numeral system, the division by powers of 2 is guaranteed to work, but $1/10$ becomes an endless binary fraction.

There's just no way to store exactly 0.1 or exactly 0.2 using the binary system, just like there is no way to store one-third as a decimal fraction.

The numeric format IEEE-754 solves this by rounding to the nearest possible number. These rounding rules normally don't allow us to see that "tiny precision loss", but it exists.

We can see this in action:

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

And when we sum two numbers, their "precision losses" add up. That's why $0.1 + 0.2$ is not exactly 0.3.

Not only JavaScript

The same issue exists in many other programming languages. PHP, Java, C, Perl, Ruby give exactly the same result, because they are based on the

same numeric format.

Can we work around the problem? Sure, the most reliable method is to round the result with the help of a method `toFixed(n)`:

```
let sum = 0.1 + 0.2;  
alert( sum.toFixed(2) ); // "0.30"
```

Please note that `toFixed` always returns a string. It ensures that it has 2 digits after the decimal point. That's actually convenient if we have an e-shopping and need to show \$0.30. For other cases, we can use the unary plus to coerce it into a number:

```
let sum = 0.1 + 0.2;  
alert( +sum.toFixed(2) ); // 0.3
```

We also can temporarily multiply the numbers by 100 (or a bigger number) to turn them into integers, do the maths, and then divide back. Then, as we're doing maths with integers, the error somewhat decreases, but we still get it on division:

```
    alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3  
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.42000000000000001
```

So, multiply/divide approach reduces the error, but doesn't remove it totally. Sometimes we could try to evade fractions at all. Like if we're dealing with a shop, then we can store prices in cents instead of dollars. But what if we apply a discount of 30%? In practice, totally evading fractions is rarely possible. Just round them to cut "tails" when needed.

The funny thing
Try running this:

```
// Hello! I'm a self-increasing number!  
alert( 9999999999999999 ); // shows 10000000000000000
```

This suffers from the same issue: a loss of precision. There are 64 bits for the number, 52 of them can be used to store digits, but that's not enough. So the least significant digits disappear. JavaScript doesn't trigger an error in such events. It does its best to fit the number into the desired format, but unfortunately, this format is not big enough.

Two zeroes

Another funny consequence of the internal representation of numbers is the existence of two zeroes: 0 and -0.

That's because a sign is represented by a single bit, so it can be set or not set for any number including a zero.

In most cases the distinction is unnoticeable, because operators are suited to treat them as the same.

Tests: `isFinite` and `isNaN` Remember these two special numeric values?

Infinity (and -Infinity) is a special numeric value that is greater (less) than anything. NaN represents an error.

They belong to the type number, but are not “normal” numbers, so there are special functions to check for them:

`isNaN(value)` converts its argument to a number and then tests it for being NaN:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

But do we need this function? Can't we just use the comparison `=== NaN`? Unfortunately not. The value NaN is unique in that it does not equal anything, including itself:

```
alert( NaN === NaN ); // false
```

isFinite(value) converts its argument to a number and returns true if it's a regular number, not NaN/Infinity/-Infinity:

```
    alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, because a special value: NaN
alert( isFinite(Infinity) ); // false, because a special value: Infinity
```

Sometimes isFinite is used to validate whether a string value is a regular number:

```
let num = +prompt("Enter a number", "");  
  
// will be true unless you enter Infinity, -Infinity or not a number  
alert( isFinite(num) );
```

Please note that an empty or a space-only string is treated as 0 in all numeric functions including `isFinite`.

`Number.isNaN` and `Number.isFinite`

`Number.isNaN` and `Number.isFinite` methods are the more “strict” versions of `isNaN` and `isFinite` functions. They do not autoconvert their argument into a number, but check if it belongs to the number type instead.

`Number.isNaN(value)` returns true if the argument belongs to the number type and it is NaN. In any other case it returns false.

```
alert( Number.isNaN(NaN) ); // true  
alert( Number.isNaN("str" / 2) ); // true
```

// Note the difference:

```
alert( Number.isNaN("str") ); // false, because "str" belongs to the string type, not the  
number type  
alert( isNaN("str") ); // true, because isNaN converts string "str" into a number and gets  
NaN as a result of this conversion
```

`Number.isFinite(value)` returns true if the argument belongs to the number type and it is not NaN/Infinity/-Infinity. In any other case it returns false.


```
    alert( Number.isFinite(123) ); // true
alert( Number.isFinite(Infinity) ); // false
alert( Number.isFinite(2 / 0) ); // false
```

// Note the difference:

```
alert( Number.isFinite("123") ); // false, because "123" belongs to the string type, not the
number type
alert( isFinite("123") ); // true, because isFinite converts string "123" into a number 123
```

In a way, `Number.isNaN` and `Number.isFinite` are simpler and more straightforward than `isNaN` and `isFinite` functions. In practice though, `isNaN` and `isFinite` are mostly used, as they're shorter to write.

Comparison with `Object.is`

There is a special built-in method `Object.is` that compares values like `===`, but is more reliable for two edge cases:

It works with NaN: `Object.is(NaN, NaN) === true`, that's a good thing.

Values 0 and -0 are different: `Object.is(0, -0) === false`, technically that's correct, because internally the number has a sign bit that may be different even if all other bits are zeroes.

In all other cases, `Object.is(a, b)` is the same as `a === b`.

We mention `Object.is` here, because it's often used in JavaScript specification. When an internal algorithm needs to compare two values for being exactly the same, it uses `Object.is` (internally called `SameValue`).

`parseInt` and `parseFloat` numeric conversion using a plus `+` or `Number()` is strict. If a value is not exactly a number, it fails:

```
alert( +"100px" ); // NaN
```

The sole exception is spaces at the beginning or at the end of the string, as they are ignored.

But in real life we often have values in units, like "100px" or "12pt" in CSS. Also in many countries the currency symbol goes after the amount, so we have "19€" and would like to extract a numeric value out of that.

That's what `parseInt` and `parseFloat` are for.

They “read” a number from a string until they can't. In case of an error, the gathered number is returned. The function `parseInt` returns an integer, whilst `parseFloat` will return a floating-point number:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5
```

```
alert( parseInt('12.3') ); // 12, only the integer part is returned
alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

There are situations when `parseInt`/`parseFloat` will return NaN. It happens when no digits could be read:

```
alert( parseInt('a123') ); // NaN, the first symbol stops the process
```

The second argument of `parseInt(str, radix)`

The `parseInt()` function has an optional second parameter. It specifies the base of the numeral system, so `parseInt` can also parse strings of hex numbers, binary numbers and so on:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, without 0x also works

alert( parseInt('2n9c', 36) ); // 123456
```

Other math functions JavaScript has a built-in `Math` object which contains a small library of mathematical functions and constants.

A few examples:

```
Math.random()
```

Returns a random number from 0 to 1 (not including 1).

```
    alert( Math.random() ); // 0.1234567894322  
alert( Math.random() ); // 0.5435252343232  
alert( Math.random() ); // ... (any random numbers)
```

`Math.max(a, b, c...)` and `Math.min(a, b, c...)`

Returns the greatest and smallest from the arbitrary number of arguments.

```
    alert( Math.max(3, 5, -10, 0, 1) ); // 5  
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

Returns n raised to the given power.

```
alert( Math.pow(2, 10) ); // 2 in power 10 = 1024
```

There are more functions and constants in Math object, including trigonometry, which you can find in the docs for the Math object.

Summary To write numbers with many zeroes:

Append "e" with the zeroes count to the number. Like: 123e6 is the same as 123 with 6 zeroes 123000000.

A negative number after "e" causes the number to be divided by 1 with given zeroes.

E.g. 123e-6 means 0.000123 (123 millionths).

For different numeral systems:

Can write numbers directly in hex (0x), octal (0o) and binary (0b) systems.

parseInt(str, base) parses the string str into an integer in numeral system with given base, 2 "d" base "d" 36.

num.toString(base) converts a number to a string in the numeral system with the given base.

For regular number tests:

isNaN(value) converts its argument to a number and then tests it for being NaN

Number.isNaN(value) checks whether its argument belongs to the number type, and if so, tests it for being NaN

isFinite(value) converts its argument to a number and then tests it for not being NaN/Infinity/-Infinity

Number.isFinite(value) checks whether its argument belongs to the number type, and if so, tests it for not being NaN/Infinity/-Infinity

For converting values like 12pt and 100px to a number:

Use `parseInt/parseFloat` for the “soft” conversion, which reads a number from a string and then returns the value they could read before the error.

For fractions:

Round using `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` or `num.toFixed(precision)`. Make sure to remember there’s a loss of precision when working with fractions.

More mathematical functions:

See the `Math` object when you need them. The library is very small, but can cover basic needs.

TasksSum numbers from the visitorimportance: 5Create a script that prompts the visitor to enter two numbers and then shows their sum.
Run the demoP.S. There is a gotcha with types.
solution

```
let a = +prompt("The first number?", "");  
let b = +prompt("The second number?", "");  
  
alert( a + b );
```

Note the unary plus `+` before `prompt`. It immediately converts the value to a number. Otherwise, `a` and `b` would be string their sum would be their concatenation, that is: `"1" + "2" = "12"`.

Why `6.35.toFixed(1) == 6.3`?importance: 4According to the documentation `Math.round` and `toFixed` both round to the nearest number: `0..4` lead down while `5..9` lead up.
For instance:

```
alert( 1.35.toFixed(1) ); // 1.4
```

In the similar example below, why is 6.35 rounded to 6.3, not 6.4?

```
alert( 6.35.toFixed(1) ); // 6.3
```

How to round 6.35 the right way?
solutionInternally the decimal fraction 6.35 is an endless binary. As always in such cases, it is stored with a precision loss.
Let's see:

```
alert( 6.35.toFixed(20) ); // 6.34999999999999964473
```

The precision loss can cause both increase and decrease of a number. In this particular case the number becomes a tiny bit less, that's why it rounded down. And what's for 1.35?

```
alert( 1.35.toFixed(20) ); // 1.35000000000000008882
```

Here the precision loss made the number a little bit greater, so it rounded up. How can we fix the problem with 6.35 if we want it to be rounded the right way? We should bring it closer to an integer prior to rounding:

```
alert( (6.35 * 10).toFixed(20) ); // 63.50000000000000000000
```

Note that 63.5 has no precision loss at all. That's because the decimal part 0.5 is actually $1/2$. Fractions divided by powers of 2 are exactly represented in the binary system, now we can round it:


```
alert( Math.round(6.35 * 10) / 10 ); // 6.35 -> 63.5 -> 64(rounded) -> 6.4
```

Repeat until the input is a numberimportance: 5Create a function readNumber which prompts for a number until the visitor enters a valid numeric value. The resulting value must be returned as a number. The visitor can also stop the process by entering an empty line or pressing "CANCEL". In that case, the function should return null. Run the demoOpen a sandbox with tests.solution

```
function readNumber() {  
  let num;  
  
  do {  
    num = prompt("Enter a number please?", 0);  
  } while ( !isFinite(num) );  
  
  if (num === null || num === "") return null;  
  
  return +num;  
}  
  
alert(`Read: ${readNumber()}`);
```

The solution is a little bit more intricate than it could be because we need to handle null/empty lines.

So we actually accept the input until it is a “regular number”. Both null (cancel) and empty line also fit that condition, because in numeric form they are 0.

After we stopped, we need to treat null and empty line specially (return null), because converting them to a number would return 0.

Open the solution with tests in a sandbox. An occasional infinite loop importance: 4 This loop is infinite. It never ends. Why?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

That's because i would never equal 10.
Run it to see the real values of i:

```
let i = 0;
while (i < 11) {
  i += 0.2;
  if (i > 9.8 && i < 10.2) alert( i );
}
```

None of them is exactly 10.
Such things happen because of the precision losses when adding fractions like 0.2.
Conclusion: evade equality checks when working with decimal fractions.
A random number from min to max importance: 2 The built-in function Math.random() creates a random value from 0 to 1 (not including 1).

Write the function `random(min, max)` to generate a random floating-point number from min to max (not including max).

Examples of its work:

```
    alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

solution We need to “map” all values from the interval $0 \dots 1$ into values from min to max.

That can be done in two stages:

If we multiply a random number from $0 \dots 1$ by $\text{max} - \text{min}$, then the interval of possible values increases $0 \dots 1$ to $0 \dots \text{max} - \text{min}$.

Now if we add min, the possible interval becomes from min to max.

The function:

```
function random(min, max) {
    return min + Math.random() * (max - min);
}
```

```
alert( random(1, 5) );
alert( random(1, 5) );
alert( random(1, 5) );
```

A random integer from min to max importance: 2 Create a function `randomInteger(min, max)` that generates a random integer number from min to max

including both min and max as possible values.

Any number from the interval min..max must appear with the same probability.

Examples of its work:

```
    alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```

You can use the solution of the previous task as the base.

The simple but wrong solutionThe simple but wrong solutionThe simplest, but wrong solution would be to generate a value from min to max and round it:

```
function randomInteger(min, max) {
  let rand = min + Math.random() * (max - min);
  return Math.round(rand);
}
```

```
alert( randomInteger(1, 3) );
```

The function works, but it is incorrect. The probability to get edge values min and max is two times less than any other.

If you run the example above many times, you would easily see that 2 appears the most often.

That happens because Math.round() gets random numbers from the interval 1..3 and rounds them as follows:

values from 1 ... to 1.4999999999 become 1
values from 1.5 ... to 2.4999999999 become 2
values from 2.5 ... to 2.9999999999 become 3

Now we can clearly see that 1 gets twice less values than 2. And the same with 3.
The correct solution
The correct solution
There are many correct solutions to the task.
One of them is to adjust interval borders. To ensure the same intervals, we can
generate values from 0.5 to 3.5, thus adding the required probabilities to the edges:

```
function randomInteger(min, max) {  
  // now rand is from (min-0.5) to (max+0.5)  
  let rand = min - 0.5 + Math.random() * (max - min + 1);  
  return Math.round(rand);  
}  
  
alert( randomInteger(1, 3) );
```

An alternative way could be to use Math.floor for a random number from min to max+1:

```

    function randomInteger(min, max) {
    // here rand is from min to (max+1)
    let rand = min + Math.random() * (max + 1 - min);
    return Math.floor(rand);
    }

    alert( randomInteger(1, 3) );

```

Now all intervals are mapped this way:

```

    values from 1 ... to 1.9999999999 become 1
    values from 2 ... to 2.9999999999 become 2
    values from 3 ... to 3.9999999999 become 3

```

All intervals have the same length, making the final distribution uniform.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var Disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://\V\javascript.info\number","identifier":"\V number"}); };var Disqus_shortcode = "javascriptinfo";var Disqus_enabled = true; TutorialThe JavaScript languageData types{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https:// javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Data types","item":"https:// javascript.info/data-types"}]}October 20, 2022StringsIn JavaScript, the textual data is stored as strings. There is no separate type for a single character. The internal format for strings is always UTF-16, it is not tied to the page encoding. QuotesLet's recall the kinds of quotes. Strings can be enclosed within either single quotes, double quotes or backticks:

```
let single = 'single-quoted';
```

```
let double = "double-quoted";
```

```
let backticks = `backticks`;
```

Single and double quotes are essentially the same. Backticks, however, allow us to embed any expression into the string, by wrapping it in `${...}`:

```
function sum(a, b) {  
  return a + b;  
}
```

```
alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Another advantage of using backticks is that they allow a string to span multiple lines:

```
let guestList = `Guests:  
* John  
* Pete  
* Mary`
```

```
`;  
`;
```

```
alert(guestList); // a list of guests, multiple lines
```

Looks natural, right? But single or double quotes do not work this way. If we use them and try to use multiple lines, there'll be an error:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL  
* John";
```

Single and double quotes come from ancient times of language creation, when the need for multiline strings was not taken into account. Backticks appeared much later and thus are more versatile.

Backticks also allow us to specify a “template function” before the first backtick. The syntax is: `func`string``. The function `func` is called automatically, receives the string and embedded expressions and can process them. This feature is called “tagged templates”, it’s rarely seen, but you can read about it in the MDN: Template literals. Special charactersIt is still possible to create multiline strings with single and double quotes by using a so-called “newline character”, written as `\n`, which denotes a line break:


```
let guestList = "Guests:\n * John\n * Pete\n * Mary";  
alert(guestList); // a multiline list of guests, same as above
```

As a simpler example, these two lines are equal, just written differently:

```
let str1 = "Hello\nWorld"; // two lines using a "newline symbol"  
  
// two lines using a normal newline and backticks  
let str2 = `Hello  
World`;  
  
alert(str1 == str2); // true
```

There are other, less common special characters:

Character	Description
-----------	-------------

<code>\n</code>	New line
-----------------	----------

<code>\r</code>	In Windows text files a combination of two characters <code>\r\n</code> represents a new break, while
-----------------	---

on non-Windows OS it's just `\n`. That's for historical reasons, most Windows software also understands `\n`.

`\', \", \``
Quotes

`\\`
Backslash

`\t`
Tab

`\b, \f, \v`
Backspace, Form Feed, Vertical Tab – mentioned for completeness, coming from old times, not used nowadays (you can forget them right now).

As you can see, all special characters start with a backslash character `\`. It is also called an “escape character”.
Because it's so special, if we need to show an actual backslash `\` within the string, we need to double it:

```
alert( `The backslash: \\` ); // The backslash: \
```

So-called “escaped” quotes `\', \", \`` are used to insert a quote into the same-quoted string.
For instance:

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

As you can see, we have to prepend the inner quote by the backslash `\`, because otherwise it would indicate the string end. Of course, only the quotes that are the same as the enclosing ones need to be escaped. So, as a more elegant solution, we could switch to double quotes or backticks instead:

```
alert( "I'm the Walrus!" ); // I'm the Walrus!
```

Besides these special characters, there's also a special notation for Unicode codes `\u...`, it's rarely used and is covered in the optional chapter about Unicode.

String lengthThe length property has the string length:

```
alert( `My\n`.length ); // 3
```

Note that `\n` is a single “special” character, so the length is indeed 3.

length is a property

People with a background in some other languages sometimes mistype by calling `str.length()` instead of just `str.length`. That doesn't work. Please note that `str.length` is a numeric property, not a function. There is no need to add parenthesis after it. Not `.length()`, but `.length`.

Accessing charactersTo get a character at position `pos`, use square brackets `[pos]` or call the method `str.at(pos)`. The first character starts from the zero position:

```
let str = `Hello`;  
  
// the first character  
alert( str[0] ); // H  
alert( str.at(0) ); // H  
  
// the last character  
alert( str[str.length - 1] ); // o  
alert( str.at(-1) );
```

As you can see, the `.at(pos)` method has a benefit of allowing negative position. If `pos` is negative, then it's counted from the end of the string. So `.at(-1)` means the last character, and `.at(-2)` is the one before it, etc.

The square brackets always return undefined for negative indexes, for instance:

```
let str = `Hello`;  
  
alert( str[-2] ); // undefined  
alert( str.at(-2) ); // l
```

We can also iterate over characters using for..of:

```
for (let char of "Hello") {  
  alert(char); // H,e,l,l,o (char becomes "H", then "e", then "l" etc)  
}
```

Strings are immutableStrings can't be changed in JavaScript. It is impossible to change a character.

Let's try it to show that it doesn't work:

```
let str = 'Hi';  
  
str[0] = 'h'; // error  
alert( str[0] ); // doesn't work
```

The usual workaround is to create a whole new string and assign it to str instead of the old one.
For instance:

```
let str = 'Hi';  
  
str = 'h' + str[1]; // replace the string  
  
alert( str ); // hi
```

In the following sections we'll see more examples of this.
Changing the caseMethods `toLowerCase()` and `toUpperCase()` change the case:

```
    alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Or, if we want a single character lowercased:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Searching for a substring There are multiple ways to look for a substring within a string.

`str.indexOf` The first method is `str.indexOf(substr, pos)`.

It looks for the `substr` in `str`, starting from the given position `pos`, and returns the position where the match was found or `-1` if nothing can be found.

For instance:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

The optional second parameter allows us to start searching from a given position. For instance, the first occurrence of "id" is at position 1. To look for the next occurrence, let's start the search from position 2:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

If we're interested in all occurrences, we can run `indexOf` in a loop. Every new call is made with the position after the previous match:

```
let str = 'As sly as a fox, as strong as an ox';
```



```

let target = 'as'; // let's look for it

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // continue the search from the next position
}

```

The same algorithm can be layed out shorter:

```

let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}

```

`str.lastIndexOf(substr, position)`

There is also a similar method `str.lastIndexOf(substr, position)` that searches from the end of a string to its beginning. It would list the occurrences in the reverse order.

There is a slight inconvenience with `indexOf` in the if test. We can't put it in the if like this:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
  alert("We found it"); // doesn't work!
}
```

The alert in the example above doesn't show because `str.indexOf("Widget")` returns 0 (meaning that it found the match at the starting position). Right, but it considers 0 to be false.

So, we should actually check for -1, like this:

```
let str = "Widget with id";

if (str.indexOf("Widget") !== -1) {
  alert("We found it"); // works now!
}
```

`includes`, `startsWith`, `endsWith` The more modern method `str.includes(substr, pos)` returns true/false depending on whether `str` contains `substr` within. It's the right choice if we need to test for the match, but don't need its position:

```
alert( "Widget with id".includes("Widget") ); // true  
alert( "Hello".includes("Bye") ); // false
```

The optional second argument of `str.includes` is the position to start searching from:

```
alert( "Widget".includes("id") ); // true  
alert( "Widget".includes("id", 3) ); // false, from position 3 there is no "id"
```

The methods `str.startsWith` and `str.endsWith` do exactly what they say:

```
    alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with "Wid"
    alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"
```

Getting a substring There are 3 methods in JavaScript to get a substring: substring, substr and slice.

```
str.slice(start [, end])
```

Returns the part of the string from start to (but not including) end.
For instance:

```
    let str = "stringify";
    alert( str.slice(0, 5) ); // 'strin', the substring from 0 to 5 (not including 5)
    alert( str.slice(0, 1) ); // 's', from 0 to 1, but not including 1, so only character at 0
```

If there is no second argument, then slice goes till the end of the string:

```
    let str = "stringify";
    alert( str.slice(2) ); // 'ringify', from the 2nd position till the end
```

Negative values for start/end are also possible. They mean the position is counted from the string end:

```
let str = "stringify";  
  
// start at the 4th position from the right, end at the 1st from the right  
alert( str.slice(-4, -1) ); // 'gif'
```

```
str.substr(start [, end])
```

Returns the part of the string between start and end (not including end).
This is almost the same as slice, but it allows start to be greater than end (in this case it simply swaps start and end values).
For instance:

```
let str = "stringify";  
  
// these are same for substring
```

```
alert( str.substring(2, 6) ); // "ring"  
alert( str.substring(6, 2) ); // "ring"
```

// ...but not for slice:

```
alert( str.slice(2, 6) ); // "ring" (the same)  
alert( str.slice(6, 2) ); // "" (an empty string)
```

Negative arguments are (unlike slice) not supported, they are treated as 0.

`str.substr(start [, length])`

Returns the part of the string from start, with the given length.

In contrast with the previous methods, this one allows us to specify the length instead of the ending position:

```
let str = "stringify";  
alert( str.substr(2, 4) ); // 'ring', from the 2nd position get 4 characters
```

The first argument may be negative, to count from the end:

```
let str = "stringify";  
alert( str.substr(-4, 2) ); // 'gi', from the 4th position get 2 characters
```

This method resides in the Annex B of the language specification. It means that only browser-hosted Javascript engines should support it, and it's not recommended to use it. In practice, it's supported everywhere.

Let's recap these methods to avoid any confusion:

method
selects...
negatives

slice(start, end)
from start to end (not including end)
allows negatives

substring(start, end)
between start and end (not including end)
negative values mean 0

substr(start, length)
from start get length characters
allows negative start

Which one to choose?

All of them can do the job. Formally, substr has a minor drawback: it is described not in the core JavaScript specification, but in Annex B, which covers browser-only features that exist mainly for historical reasons. So, non-browser environments may fail to support it. But in practice it works everywhere. Of the other two variants, slice is a little bit more flexible, it allows negative arguments and shorter to write. So, for practical use it's enough to remember only slice.

Comparing stringsAs we know from the chapter Comparisons, strings are compared character-by-character in alphabetical order. Although, there are some oddities.

A lowercase letter is always greater than the uppercase:

```
alert( 'a' > 'Z' ); // true
```

Letters with diacritical marks are “out of order”:

```
alert( 'Österreich' > 'Zealand' ); // true
```

This may lead to strange results if we sort these country names. Usually people would expect Zealand to come after Österreich in the list.

To understand what happens, we should be aware that strings in Javascript are encoded using UTF-16. That is: each character has a corresponding numeric code. There are special methods that allow to get the character for the code and back:

`str.codePointAt(pos)`

Returns a decimal number representing the code for the character at position pos:

```
// different case letters have different codes
alert( "Z".codePointAt(0) ); // 90
alert( "z".codePointAt(0) ); // 122
alert( "z".codePointAt(0).toString(16) ); // 7a (if we need a hexadecimal value)
```

`String.fromCodePoint(code)`

Creates a character by its numeric code

```
alert( String.fromCodePoint(90) ); // Z
alert( String.fromCodePoint(0x5a) ); // Z (we can also use a hex value as an argument)
```

Now let's see the characters with codes 65..220 (the latin alphabet and a little bit extra) by making a string of them:

```
let str = "";

for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
alert( str );
// Output:
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~•€•,f,,
// ¡¢£¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```

See? Capital characters go first, then a few special ones, then lowercase characters, and Õ near the end of the output. Now it becomes obvious why a > Z. The characters are compared by their numeric code. The greater code means that the character is greater. The code for a (97) is greater than the code for Z (90).

All lowercase letters go after uppercase letters because their codes are greater. Some letters like Õ stand apart from the main alphabet. Here, its code is greater than anything from a to z.

Correct comparisonsThe “right” algorithm to do string comparisons is more complex than it may seem, because alphabets are different for different languages.

So, the browser needs to know the language to compare.

Luckily, modern browsers support the internationalization standard ECMA-402.

It provides a special method to compare strings in different languages, following their rules.

The call `str.localeCompare(str2)` returns an integer indicating whether str is less, equal

or greater than str2 according to the language rules:

Returns a negative number if str is less than str2.

Returns a positive number if str is greater than str2.

Returns 0 if they are equivalent.

For instance:

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

This method actually has two additional arguments specified in the documentation, which allows it to specify the language (by default taken from the environment, letter order depends on the language) and setup additional rules like case sensitivity or should "a" and "a0 " &R G&V FVB 2 F†R 6 ÖR WF2à

Summary

There are 3 types of quotes. Backticks allow a string to span multiple lines and embed expressions \${...}.

We can use special characters, such as a line break \n.

To get a character, use: [] or at method.

To get a substring, use: slice or substring.

To lowercase/uppercase a string, use: toLowerCase/toUpperCase.

To look for a substring, use: indexOf, or includes/startsWith/endsWith for simple checks.

To compare strings according to the language, use: localeCompare, otherwise they are compared by character codes.

There are several other helpful methods in strings:

str.trim() – removes (“trims”) spaces from the beginning and end of the string.

str.repeat(n) – repeats the string n times.

...and more to be found in the manual.

Strings also have methods for doing search/replace with regular expressions. But that's big topic, so it's explained in a separate tutorial section Regular expressions.

Also, as of now it's important to know that strings are based on Unicode encoding, and hence there're issues with comparisons. There's more about Unicode in the chapter Unicode, String internals.

Tasks
Uppercase the first character
importance: 5
Write a function `ucFirst(str)` that returns the string `str` with the uppercased first character, for instance:

```
ucFirst("john") == "John";
```

Open a sandbox with tests.
solution
We can't "replace" the first character, because strings in JavaScript are immutable.

But we can make a new string based on the existing one, with the uppercased first character:

```
let newStr = str[0].toUpperCase() + str.slice(1);
```

There's a small problem though. If `str` is empty, then `str[0]` is undefined, and as undefined doesn't have the `toUpperCase()` method, we'll get an error. The easiest way out is to add a test for an empty string, like this:

```
function ucFirst(str) {  
  if (!str) return str;  
  
  return str[0].toUpperCase() + str.slice(1);  
}
```

```
alert( ucFirst("john") ); // John
```

Open the solution with tests in a sandbox. Check for spam importance: 5 Write a function `checkSpam(str)` that returns `true` if `str` contains 'viagra' or 'XXX', otherwise `false`.

The function must be case-insensitive:

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxxx') == true
checkSpam("innocent rabbit") == false
```

Open a sandbox with tests. solution To make the search case-insensitive, let's bring the string to lower case and then search:

```
function checkSpam(str) {
  let lowerStr = str.toLowerCase();

  return lowerStr.includes('viagra') || lowerStr.includes('xxx');
}

alert( checkSpam('buy ViAgRA now') );
alert( checkSpam('free xxxxx') );
alert( checkSpam("innocent rabbit") );
```

Open the solution with tests in a sandbox. Truncate the text importance: 5 Create a function `truncate(str, maxLength)` that checks the length of the `str` and, if it exceeds

maxlength – replaces the end of str with the ellipsis character "...", to make its length equal to maxlength.

The result of the function should be the truncated (if needed) string.

For instance:

```
truncate("What I'd like to tell on this topic is:", 20) = "What I'd like to te..."
```

```
truncate("Hi everyone!", 20) = "Hi everyone!"
```

Open a sandbox with tests.
The maximal length must be maxlength, so we need to cut it a little shorter, to give space for the ellipsis.

Note that there is actually a single Unicode character for an ellipsis. That's not three dots.

```
function truncate(str, maxlength) {  
  return (str.length > maxlength) ?  
    str.slice(0, maxlength - 1) + '...' : str;  
}
```

Open the solution with tests in a sandbox.
Extract the money importance: 4
We have a cost in the form "\$120". That is: the dollar sign goes first, and then the number.

Create a function extractCurrencyValue(str) that would extract the numeric value from such string and return it.

The example:

```
alert( extractCurrencyValue('$120') === 120 ); // true
```

Open a sandbox with tests.solution

```
function extractCurrencyValue(str) {  
  return +str.slice(1);  
}
```

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info","identifier":"string"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},{"@type":"ListItem","position":3,"name":"Data types","item":"https://javascript.info/data-types"}]}January 24, 2023ArraysObjects allow you to store keyed collections of values. That's fine.

But quite often we find that we need an ordered collection, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones.

Objects are just not meant for such use.

There exists a special data structure named Array, to store ordered collections.

DeclarationThere are two syntaxes for creating an empty array:

```
let arr = new Array();  
let arr = [];
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Array elements are numbered, starting with zero.
We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...Or add a new one to the array:


```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

The total count of the elements in the array is its length:

```
let fruits = ["Apple", "Orange", "Plum"];  
alert( fruits.length ); // 3
```

We can also use alert to show the whole array.

```
let fruits = ["Apple", "Orange", "Plum"];  
alert( fruits ); // Apple,Orange,Plum
```

An array can store elements of any type.
For instance:

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John

// get the function at index 3 and run it
arr[3]() // hello
```

Trailing comma

An array, just like an object, may end with a comma:

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

The “trailing comma” style makes it easier to insert/remove items, because all lines become alike.

Get last elements with “at”

A recent addition

This is a recent addition to the language.
Old browsers may need polyfills.

Let's say we want the last element of the array.

Some programming languages allow the use of negative indexes for the same purpose, like `fruits[-1]`.

Although, in JavaScript it won't work. The result will be undefined, because the index in square brackets is treated literally.

We can explicitly calculate the last element index and then access it: `fruits[fruits.length - 1]`.

```
let fruits = ["Apple", "Orange", "Plum"];  
  
alert( fruits[fruits.length-1] ); // Plum
```

A bit cumbersome, isn't it? We need to write the variable name twice. Luckily, there's a shorter syntax: `fruits.at(-1)`:

```
let fruits = ["Apple", "Orange", "Plum"];  
  
// same as fruits[fruits.length-1]  
alert( fruits.at(-1) ); // Plum
```

In other words, `arr.at(i)`:

is exactly the same as `arr[i]`, if $i \geq 0$.

for negative values of i , it steps back from the end of the array.

Methods `pop`/`push`, `shift`/`unshift` A queue is one of the most common uses of an array. In computer science, this means an ordered collection of elements which supports two operations:

`push` appends an element to the end.

`shift` get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

Arrays support both operations.

In practice we need it very often. For example, a queue of messages that need to be shown on-screen.

There's another use case for arrays – the data structure named stack.

It supports two operations:

`push` adds an element to the end.

`pop` takes an element from the end.

So new elements are added or taken always from the “end”.

A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:

For stacks, the latest pushed item is received first, that's also called LIFO (Last-In-First-Out) principle. For queues, we have FIFO (First-In-First-Out).

Arrays in JavaScript can work both as a queue and as a stack. They allow you to add/remove elements, both to/from the beginning or the end.

In computer science, the data structure that allows this, is called deque.

Methods that work with the end of the array:

`pop`

Extracts the last element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];  
alert( fruits.pop() ); // remove "Pear" and alert it  
alert( fruits ); // Apple, Orange
```

Both `fruits.pop()` and `fruits.at(-1)` return the last element of the array, but `fruits.pop()` also modifies the array by removing it.

push

Append the element to the end of the array:

```
let fruits = ["Apple", "Orange"];  
fruits.push("Pear");  
alert( fruits ); // Apple, Orange, Pear
```

The call `fruits.push(...)` is equal to `fruits[fruits.length] =`

Methods that work with the beginning of the array:

shift

Extracts the first element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];  
alert( fruits.shift() ); // remove Apple and alert it  
alert( fruits ); // Orange, Pear
```

unshift

Add the element to the beginning of the array:

```
let fruits = ["Orange", "Pear"];  
fruits.unshift('Apple');
```

```
alert( fruits ); // Apple, Orange, Pear
```

Methods push and unshift can add multiple elements at once:

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

InternalsAn array is a special kind of object. The square brackets used to access a property `arr[0]` actually come from the object syntax. That's essentially the same as `obj[key]`, where `arr` is the object, while numbers are used as keys. They extend objects providing special methods to work with ordered collections of data and also the `length` property. But at the core it's still an object. Remember, there are only eight basic data types in JavaScript (see the Data types chapter for more info). Array is an object and thus behaves like an object. For instance, it is copied by reference:

```
let fruits = ["Banana"]  
  
let arr = fruits; // copy by reference (two variables reference the same array)  
  
alert( arr === fruits ); // true  
  
arr.push("Pear"); // modify the array by reference  
  
alert( fruits ); // Banana, Pear - 2 items now
```

...But what makes arrays really special is their internal representation. The engine tries to store its elements in the contiguous memory area, one after another, just as depicted on the illustrations in this chapter, and there are other optimizations as well, to make arrays work really fast.

But they all break if we quit working with an array as with an “ordered collection” and start working with it as if it were a regular object.

For instance, technically we can do this:

```
let fruits = []; // make an array  
  
fruits[999999] = 5; // assign a property with the index far greater than its length  
  
fruits.age = 25; // create a property with an arbitrary name
```

That's possible, because arrays are objects at their base. We can add any properties to them.

But the engine will see that we're working with the array as with a regular object. Array-specific optimizations are not suited for such cases and will be turned off, their benefits disappear.

The ways to misuse an array:

Add a non-numeric property like `arr.test = 5`.

Make holes, like: add `arr[0]` and then `arr[1000]` (and nothing between them).

Fill the array in the reverse order, like `arr[1000]`, `arr[999]` and so on.

Please think of arrays as special structures to work with the ordered data. They provide special methods for that. Arrays are carefully tuned inside JavaScript engines to work with contiguous ordered data, please use them this way. And if you need arbitrary keys, chances are high that you actually require a regular object {}.

PerformanceMethods push/pop run fast, while shift/unshift are slow.

Why is it faster to work with the end of an array than with its beginning? Let's see what happens during the execution:

```
fruits.shift(); // take 1 element from the start
```

It's not enough to take and remove the element with the index 0. Other elements need to be renumbered as well.

The shift operation must do 3 things:

Remove the element with the index 0.

Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on.

Update the length property.

The more elements in the array, the more time to move them, more in-memory operations.

The similar thing happens with unshift: to add an element to the beginning of the array, we need first to move existing elements to the right, increasing their indexes.

And what's with push/pop? They do not need to move anything. To extract an element from the end, the pop method cleans the index and shortens length.

The actions for the pop operation:

```
fruits.pop(); // take 1 element from the end
```

The pop method does not need to move anything, because other elements keep their indexes. That's why it's blazingly fast.

The similar thing with the push method.

Loops One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

But for arrays there is another form of loop, for..of:

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

The `for..of` doesn't give access to the number of the current element, just its value, but in most cases that's enough. And it's shorter.

Technically, because arrays are objects, it is also possible to use `for..in`:

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

But that's actually a bad idea. There are potential problems with it:

The loop `for..in` iterates over all properties, not only the numeric ones.

There are so-called “array-like” objects in the browser and in other environments, that look like arrays. That is, they have `length` and `indexes` properties, but they may also have other non-numeric properties and methods, which we usually don't need. The `for..in` loop will list them though. So if we need to work with array-like objects, then these “extra” properties can become a problem.

The `for..in` loop is optimized for generic objects, not arrays, and thus is 10-100 times slower. Of course, it's still very fast. The speedup may only matter in bottlenecks. But still we should be aware of the difference.

Generally, we shouldn't use `for..in` for arrays.

A word about "length" The length property automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but the greatest numeric index plus one.

For instance, a single element with a large index gives a big length:

```
let fruits = [];  
fruits[123] = "Apple";  
  
alert( fruits.length ); // 124
```

Note that we usually don't use arrays like that.

Another interesting thing about the length property is that it's writable.

If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here's the example:

```
let arr = [1, 2, 3, 4, 5];  
  
arr.length = 2; // truncate to 2 elements  
alert( arr ); // [1, 2]
```

```
arr.length = 5; // return length back  
alert( arr[3] ); // undefined: the values do not return
```

So, the simplest way to clear the array is: `arr.length = 0;`
`new Array()` There is one more syntax to create an array:

```
let arr = new Array("Apple", "Pear", "etc");
```

It's rarely used, because square brackets `[]` are shorter. Also, there's a tricky feature with it.

If `new Array` is called with a single argument which is a number, then it creates an array without items, but with the given length.

Let's see how one can shoot themselves in the foot:

```
let arr = new Array(2); // will it create an array of [2] ?  
  
alert( arr[0] ); // undefined! no elements.  
  
alert( arr.length ); // length 2
```

To avoid such surprises, we usually use square brackets, unless we really know what we're doing.

Multidimensional arrays Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices:

```
let matrix = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];  
  
alert( matrix[1][1] ); // 5, the central element
```

toStringArrays have their own implementation of toString method that returns a comma-separated list of elements.
For instance:

```
let arr = [1, 2, 3];  
  
alert( arr ); // 1,2,3  
alert( String(arr) === '1,2,3' ); // true
```

Also, let's try this:

```
    alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Arrays do not have `Symbol.toPrimitive`, neither a viable `valueOf`, they implement only `toString` conversion, so here `[]` becomes an empty string, `[1]` becomes `"1"` and `[1,2]` becomes `"1,2"`.

When the binary plus `+` operator adds something to a string, it converts it to a string as well, so the next step looks like this:

```
    alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

Don't compare arrays with `==`. Arrays in JavaScript, unlike some other programming languages, shouldn't be compared with operator `==`.

This operator has no special treatment for arrays, it works with them as with any objects. Let's recall the rules:

Two objects are equal `==` only if they're references to the same object.

If one of the arguments of `==` is an object, and the other one is a primitive, then the object gets converted to primitive, as explained in the chapter Object to primitive conversion.

...With an exception of null and undefined that equal == each other and nothing else.

The strict comparison === is even simpler, as it doesn't convert types.

So, if we compare arrays with ==, they are never the same, unless we compare two variables that reference exactly the same array.

For example:

```
    alert( [] == [] ); // false
alert( [0] == [0] ); // false
```

These arrays are technically different objects. So they aren't equal. The == operator doesn't do item-by-item comparison.

Comparison with primitives may give seemingly strange results as well:

```
    alert( 0 == [] ); // true
alert( '0' == [] ); // false
```

Here, in both cases, we compare a primitive with an array object. So the array [] gets converted to primitive for the purpose of comparison and becomes an empty string "".

Then the comparison process goes on with the primitives, as described in the chapter Type Conversions:

```
// after [] was converted to "  
alert( 0 == " ); // true, as " becomes converted to number 0  
  
alert('0' == " ); // false, no type conversion, different strings
```

So, how to compare arrays?

That's simple: don't use the == operator. Instead, compare them item-by-item in a loop or using iteration methods explained in the next chapter.

SummaryArray is a special kind of object, suited to storing and managing ordered data items.

The declaration:

```
// square brackets (usual)  
let arr = [item1, item2...];  
  
// new Array (exceptionally rare)  
let arr = new Array(item1, item2...);
```

The call to new Array(number) creates an array with the given length, but without elements.

The length property is the array length or, to be precise, its last numeric index plus one. It is auto-adjusted by array methods.

If we shorten length manually, the array is truncated.

Getting the elements:

we can get element by its index, like `arr[0]`

also we can use `at(i)` method that allows negative indexes. For negative values of `i`, it steps back from the end of the array. If `i >= 0`, it works same as `arr[i]`.

We can use an array as a deque with the following operations:

`push(...items)` adds items to the end.

`pop()` removes the element from the end and returns it.

`shift()` removes the element from the beginning and returns it.

`unshift(...items)` adds items to the beginning.

To loop over the elements of the array:

`for (let i=0; i<arr.length; i++)` – works fastest, old-browser-compatible.

`for (let item of arr)` – the modern syntax for items only,

`for (let i in arr)` – never use.

To compare arrays, don't use the `==` operator (as well as `>`, `<` and others), as they have no special treatment for arrays. They handle them as any objects, and it's not what we usually want.

Instead you can use `for..of` loop to compare arrays item-by-item.

We will continue with arrays and study more methods to add, remove, extract elements and sort arrays in the next chapter Array methods.

TasksIs array copied?importance: 3What is this code going to show?

```
let fruits = ["Apples", "Pear", "Orange"];

// push a new value into the "copy"
let shoppingCart = fruits;
shoppingCart.push("Banana");

// what's in fruits?
alert( fruits.length ); // ?
```

solutionThe result is 4:

```
let fruits = ["Apples", "Pear", "Orange"];

let shoppingCart = fruits;

shoppingCart.push("Banana");

alert( fruits.length ); // 4
```

That's because arrays are objects. So both shoppingCart and fruits are the references to the same array.

Array operations. importance: 5 Let's try 5 array operations.

Create an array styles with items "Jazz" and "Blues".

Append "Rock-n-Roll" to the end.

Replace the value in the middle with "Classics". Your code for finding the middle value should work for any arrays with odd length.

Strip off the first value of the array and show it.

Prepend Rap and Reggae to the array.

The array in the process:

```
Jazz, Blues
Jazz, Blues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

solution

```
let styles = ["Jazz", "Blues"];
styles.push("Rock-n-Roll");
styles[Math.floor((styles.length - 1) / 2)] = "Classics";
alert( styles.shift() );
styles.unshift("Rap", "Reggae");
```

Calling in an array context importance: 5 What is the result? Why?

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
});

arr[2](); // ?
```

solution The call `arr[2]()` is syntactically the good old `obj[method]()`, in the role of `obj` we have `arr`, and in the role of `method` we have `2`.
So we have a call of the function `arr[2]` as an object method. Naturally, it receives this referencing the object `arr` and outputs the array:

```

    let arr = ["a", "b"];

arr.push(function() {
    alert( this );
})

arr[2](); // a,b,function(){...}

```

The array has 3 values: initially it had two, plus the function.
 Sum input numbersimportance: 4Write the function sumInput() that:

Asks the user for values using prompt and stores the values in the array.
 Finishes asking when the user enters a non-numeric value, an empty string, or presses
 "Cancel".
 Calculates and returns the sum of array items.

P.S. A zero 0 is a valid number, please don't stop the input on zero.
 Run the demosolutionPlease note the subtle, but important detail of the solution. We
 don't convert value to number instantly after prompt, because after value = +value we
 would not be able to tell an empty string (stop sign) from the zero (valid number). We do
 it later instead.

```

function sumInput() {

let numbers = [];

while (true) {

    let value = prompt("A number please?", 0);

    // should we cancel?
    if (value === "" || value === null || !isFinite(value)) break;

```

```

    numbers.push(+value);
  }

  let sum = 0;
  for (let number of numbers) {
    sum += number;
  }
  return sum;
}

alert( sumInput() );

```

A maximal subarray importance: 2The input is an array of numbers, e.g. `arr = [1, -2, 3, 4, -9, 6]`.
 The task is: find the contiguous subarray of `arr` with the maximal sum of items.
 Write the function `getMaxSubSum(arr)` that will return that sum.
 For instance:

```

getMaxSubSum([-1, 2, 3, -9]) == 5 (the sum of highlighted items)
getMaxSubSum([2, -1, 2, 3, -9]) == 6
getMaxSubSum([-1, 2, 3, -9, 11]) == 11
getMaxSubSum([-2, -1, 1, 2]) == 3
getMaxSubSum([100, -9, 2, -3, 5]) == 100
getMaxSubSum([1, 2, 3]) == 6 (take all)

```

If all items are negative, it means that we take none (the subarray is empty), so the sum is zero:

```

getMaxSubSum([-1, -2, -3]) = 0

```

Please try to think of a fast solution: $O(n^2)$ or even $O(n)$ if you can.
 Open a sandbox with tests.
 solutionSlow solutionSlow solutionWe can calculate all possible subsums.

The simplest way is to take every element and calculate sums of all subarrays starting from it.

For instance, for $[-1, 2, 3, -9, 11]$:

// Starting from -1:

-1

-1 + 2

-1 + 2 + 3

-1 + 2 + 3 + (-9)

-1 + 2 + 3 + (-9) + 11

// Starting from 2:

2

2 + 3

2 + 3 + (-9)

2 + 3 + (-9) + 11

// Starting from 3:

3

3 + (-9)

3 + (-9) + 11

// Starting from -9

-9

-9 + 11

// Starting from 11

11

The code is actually a nested loop: the external loop over array elements, and the internal counts subsums starting with the current element.

```

function getMaxSubSum(arr) {
let maxSum = 0; // if we take no elements, zero will be returned

for (let i = 0; i < arr.length; i++) {
  let sumFixedStart = 0;
  for (let j = i; j < arr.length; j++) {
    sumFixedStart += arr[j];
    maxSum = Math.max(maxSum, sumFixedStart);
  }
}

return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100

```

The solution has a time complexity of $O(n^2)$. In other words, if we increase the array size 2 times, the algorithm will work 4 times longer.

For big arrays (1000, 10000 or more items) such algorithms can lead to serious sluggishness.

Fast solution Let's walk the array and keep the current partial sum of elements in the variable *s*. If *s* becomes negative at some point, then assign *s*=0. The maximum of all such *s* will be the answer.

If the description is too vague, please see the code, it's short enough:

```

function getMaxSubSum(arr) {
let maxSum = 0;

```



```

let partialSum = 0;

for (let item of arr) { // for each item of arr
  partialSum += item; // add it to partialSum
  maxSum = Math.max(maxSum, partialSum); // remember the maximum
  if (partialSum < 0) partialSum = 0; // zero if negative
}

return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([-1, -2, -3]) ); // 0

```

The algorithm requires exactly 1 array pass, so the time complexity is $O(n)$. You can find more detailed information about the algorithm here: [Maximum subarray problem](#). If it's still not obvious why that works, then please trace the algorithm on the examples above, see how it works, that's better than any words.

Open the solution with tests in a [sandbox](#). Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a [GitHub issue](#) or a [pull request](#) instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a [sandbox](#) (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/Varray","identifier":"Varray"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Data types","item":"https://javascript.info/data-types"}]}December 6, 2022Array methodsArrays provide a lot of methods. To make things easier, in this chapter they are split into groups.

Add/remove itemsWe already know methods that add and remove items from the beginning or the end:

```

arr.push(...items) – adds items to the end,
arr.pop() – extracts an item from the end,
arr.shift() – extracts an item from the beginning,

```

`arr.unshift(...items)` – adds items to the beginning.

Here are a few others.

`splice` How to delete an element from the array?

The arrays are objects, so we can try to use `delete`:

```
let arr = ["I", "go", "home"];

delete arr[1]; // remove "go"

alert( arr[1] ); // undefined

// now arr = ["I", , "home"];
alert( arr.length ); // 3
```

The element was removed, but the array still has 3 elements, we can see that `arr.length == 3`.

That's natural, because `delete obj.key` removes a value by the key. It's all it does. Fine for objects. But for arrays we usually want the rest of elements to shift and occupy the freed place. We expect to have a shorter array now.

So, special methods should be used.

The `arr.splice` method is a swiss army knife for arrays. It can do everything: insert, remove and replace elements.

The syntax is:

```
arr.splice(start[, deleteCount, elem1, ..., elemN])
```

It modifies `arr` starting from the index `start`: removes `deleteCount` elements and then

inserts elem1, ..., elemN at their place. Returns the array of removed elements.
This method is easy to grasp by examples.
Let's start with the deletion:

```
let arr = ["I", "study", "JavaScript"];  
  
arr.splice(1, 1); // from index 1 remove 1 element  
  
alert( arr ); // ["I", "JavaScript"]
```

Easy, right? Starting from the index 1 it removed 1 element.
In the next example we remove 3 elements and replace them with the other two:

```
let arr = ["I", "study", "JavaScript", "right", "now"];  
  
// remove 3 first elements and replace them with another  
arr.splice(0, 3, "Let's", "dance");  
  
alert( arr ) // now ["Let's", "dance", "right", "now"]
```

Here we can see that splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 2 first elements
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- array of removed elements
```

The splice method is also able to insert the elements without any removals. For that we need to set deleteCount to 0:

```
let arr = ["I", "study", "JavaScript"];

// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

Negative indexes allowed

Here and in other array methods, negative indexes are allowed. They specify the position from the end of the array, like here:

```
let arr = [1, 2, 5];  
  
// from index -1 (one step from the end)  
// delete 0 elements,  
// then insert 3 and 4  
arr.splice(-1, 0, 3, 4);  
  
alert( arr ); // 1,2,3,4,5
```

sliceThe method `arr.slice` is much simpler than similar-looking `arr.splice`.
The syntax is:

```
arr.slice([start], [end])
```

It returns a new array copying to it all items from index `start` to `end` (not including `end`). Both `start` and `end` can be negative, in that case position from array end is assumed.

It's similar to a string method `str.slice`, but instead of substrings it makes subarrays.
For instance:

```
let arr = ["t", "e", "s", "t"];  
  
alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)  
  
alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

We can also call it without arguments: `arr.slice()` creates a copy of `arr`. That's often used to obtain a copy for further transformations that should not affect the original array.

The method `arr.concat` creates a new array that includes values from other arrays and additional items.

The syntax is:

```
arr.concat(arg1, arg2...)
```

It accepts any number of arguments – either arrays or values.

The result is a new array containing items from `arr`, then `arg1`, `arg2` etc.

If an argument `argN` is an array, then all its elements are copied. Otherwise, the argument itself is copied.

For instance:

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Normally, it only copies elements from arrays. Other objects, even if they look like arrays, are added as a whole:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

...But if an array-like object has a special `Symbol.isConcatSpreadable` property, then it's treated as an array by `concat`: its elements are added instead:

```
    let arr = [1, 2];

let arrayLike = {
  0: "something",
  1: "else",
  [Symbol.isConcatSpreadable]: true,
  length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,something,else
```

Iterate: forEach The arr.forEach method allows to run a function for every element of the array.
The syntax:

```
    arr.forEach(function(item, index, array) {
// ... do something with item
});
```

For instance, this shows each element of the array:


```
// for each element call alert  
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

And this code is more elaborate about their positions in the target array:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {  
  alert(` ${item} is at index ${index} in ${array}`);  
});
```

The result of the function (if it returns any) is thrown away and ignored.
Searching in arrayNow let's cover methods that search in an array.
indexOf/lastIndexOf and includesThe methods `arr.indexOf` and `arr.includes` have the similar syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

`arr.indexOf(item, from)` – looks for item starting from index from, and returns the index where it was found, otherwise -1.

`arr.includes(item, from)` – looks for item starting from index from, returns true if found.

Usually these methods are used with only one argument: the item to search. By default, the search is from the beginning.

For instance:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

Please note that `indexOf` uses the strict equality `===` for comparison. So, if we look for `false`, it finds exactly `false` and not the zero.

If we want to check if item exists in the array, and don't need the index, then `arr.includes` is preferred.

The method `arr.lastIndexOf` is the same as `indexOf`, but looks for from right to left.

```
let fruits = ['Apple', 'Orange', 'Apple']

alert( fruits.indexOf('Apple') ); // 0 (first Apple)
alert( fruits.lastIndexOf('Apple') ); // 2 (last Apple)
```

The `includes` method handles `NaN` correctly

A minor, but noteworthy feature of `includes` is that it correctly handles `NaN`, unlike `indexOf`:

```
const arr = [NaN];  
alert( arr.indexOf(NaN) ); // -1 (wrong, should be 0)  
alert( arr.includes(NaN) ); // true (correct)
```

That's because includes was added to JavaScript much later and uses the more up to date comparison algorithm internally.

find and findIndex/findLastIndexImagine we have an array of objects. How do we find an object with the specific condition?

Here the arr.find(fn) method comes in handy.

The syntax is:

```
let result = arr.find(function(item, index, array) {  
  // if true is returned, item is returned and iteration is stopped  
  // for falsy scenario returns undefined  
});
```

The function is called for elements of the array, one after another:

item is the element.

index is its index.

array is the array itself.

If it returns true, the search is stopped, the item is returned. If nothing found, undefined is returned.

For example, we have an array of users, each with the fields id and name. Let's find the one with id == 1:

```
    let users = [  
      {id: 1, name: "John"},  
      {id: 2, name: "Pete"},  
      {id: 3, name: "Mary"}  
    ];  
  
    let user = users.find(item => item.id == 1);  
  
    alert(user.name); // John
```

In real life arrays of objects is a common thing, so the find method is very useful. Note that in the example we provide to find the function `item => item.id == 1` with one argument. That's typical, other arguments of this function are rarely used. The `arr.findIndex` method has the same syntax, but returns the index where the element was found instead of the element itself. The value of -1 is returned if nothing is found. The `arr.findLastIndex` method is like `findIndex`, but searches from right to left, similar to `lastIndexOf`. Here's an example:

```
    let users = [  
      {id: 1, name: "John"},  
      {id: 2, name: "Pete"},  
      {id: 3, name: "Mary"},  
      {id: 4, name: "John"}  
    ];
```

```
// Find the index of the first John
alert(users.findIndex(user => user.name == 'John')); // 0
```

```
// Find the index of the last John
alert(users.findLastIndex(user => user.name == 'John')); // 3
```

filterThe find method looks for a single (first) element that makes the function return true.

If there may be many, we can use `arr.filter(fn)`.

The syntax is similar to find, but filter returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
```

For instance:

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);
```

```
alert(someUsers.length); // 2
```

Transform an array
Let's move on to methods that transform and reorder an array.
The `arr.map` method is one of the most useful and often used.
It calls the function for each element of the array and returns the array of results.
The syntax is:

```
let result = arr.map(function(item, index, array) {  
  // returns the new value instead of item  
});
```

For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);  
alert(lengths); // 5,7,6
```

`sort(fn)`
The call to `arr.sort()` sorts the array in place, changing its element order.
It also returns the sorted array, but the returned value is usually ignored, as `arr` itself is modified.
For instance:

```
let arr = [ 1, 2, 15 ];  
  
// the method reorders the content of arr  
arr.sort();  
  
alert( arr ); // 1, 15, 2
```

Did you notice anything strange in the outcome?
The order became 1, 15, 2. Incorrect. But why?
The items are sorted as strings by default.
Literally, all elements are converted to strings for comparisons. For strings, lexicographic ordering is applied and indeed "2" > "15".
To use our own sorting order, we need to supply a function as the argument of arr.sort().
The function should compare two arbitrary values and return:

```
function compare(a, b) {  
  if (a > b) return 1; // if the first value is greater than the second  
  if (a == b) return 0; // if values are equal  
  if (a < b) return -1; // if the first value is less than the second  
}
```

For instance, to sort as numbers:

```
function compareNumeric(a, b) {  
  if (a > b) return 1;  
  if (a == b) return 0;  
  if (a < b) return -1;  
}
```

```
let arr = [ 1, 2, 15 ];
```

```
arr.sort(compareNumeric);
```

```
alert(arr); // 1, 2, 15
```

Now it works as intended.

Let's step aside and think what's happening. The arr can be array of anything, right? It may contain numbers or strings or objects or whatever. We have a set of some items. To sort it, we need an ordering function that knows how to compare its elements. The default is a string order.

The arr.sort(fn) method implements a generic sorting algorithm. We don't need to care how it internally works (an optimized quicksort or Timsort most of the time). It will walk the array, compare its elements using the provided function and reorder them, all we need is to provide the fn which does the comparison.

By the way, if we ever want to know which elements are compared – nothing prevents from alerting them:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {  
  alert( a + " <> " + b );  
  return a - b;  
});
```

The algorithm may compare an element with multiple others in the process, but it

tries to make as few comparisons as possible.

A comparison function may return any number

Actually, a comparison function is only required to return a positive number to say “greater” and a negative number to say “less”.

That allows to write shorter functions:

```
let arr = [ 1, 2, 15 ];  
arr.sort(function(a, b) { return a - b; });  
alert(arr); // 1, 2, 15
```

Arrow functions for the best

Remember arrow functions? We can use them here for neater sorting:

```
arr.sort( (a, b) => a - b );
```

This works exactly the same as the longer version above.

Use localeCompare for strings

Remember strings comparison algorithm? It compares letters by their codes by default.

For many alphabets, it's better to use str.localeCompare method to correctly sort letters, such as Ö.

For example, let's sort a few countries in German:

```
let countries = ['Österreich', 'Andorra', 'Vietnam'];  
  
alert( countries.sort( (a, b) => a > b ? 1 : -1 ) ); // Andorra, Vietnam, Österreich (wrong)  
  
alert( countries.sort( (a, b) => a.localeCompare(b) ) ); // Andorra, Österreich, Vietnam  
(correct!)
```

reverseThe method `arr.reverse` reverses the order of elements in `arr`.
For instance:

```
let arr = [1, 2, 3, 4, 5];  
arr.reverse();  
  
alert( arr ); // 5,4,3,2,1
```

It also returns the array `arr` after the reversal.
split and joinHere's the situation from real life. We are writing a messaging app, and the

person enters the comma-delimited list of receivers: John, Pete, Mary. But for us an array of names would be much more comfortable than a single string. How to get it? The `str.split(delim)` method does exactly that. It splits the string into an array by the given delimiter `delim`.

In the example below, we split by a comma followed by space:

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
  alert( `A message to ${name}.` ); // A message to Bilbo (and other names)
}
```

The `split` method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

Split into letters

The call to `split(s)` with an empty `s` would split the string into an array of letters:

```
let str = "test";  
alert( str.split("") ); // t,e,s,t
```

The call `arr.join(glue)` does the reverse to `split`. It creates a string of `arr` items joined by `glue` between them.

For instance:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];  
let str = arr.join(';'); // glue the array into a string using ;  
alert( str ); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight When we need to iterate over an array – we can use forEach, for or for..of.

When we need to iterate and return the data for each element – we can use map.

The methods arr.reduce and arr.reduceRight also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

The syntax is:

```
let value = arr.reduce(function(accumulator, item, index, array) {  
  // ...  
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

Arguments:

accumulator – is the result of the previous function call, equals initial the first time (if initial is provided).

item – is the current array item.

index – is its position.

array – is the array.

As function is applied, the result of the previous function call is passed to the next one as the first argument.

So, the first argument is essentially the accumulator that stores the combined result of all previous executions. And at the end it becomes the result of reduce.

Sounds complicated?

The easiest way to grasp that is by example.

Here we get a sum of an array in one line:

```
let arr = [1, 2, 3, 4, 5];  
  
let result = arr.reduce((sum, current) => sum + current, 0);  
  
alert(result); // 15
```

The function passed to reduce uses only 2 arguments, that's typically enough. Let's see the details of what's going on.

On the first run, sum is the initial value (the last argument of reduce), equals 0, and current is the first array element, equals 1. So the function result is 1.
On the second run, sum = 1, we add the second array element (2) to it and return.
On the 3rd run, sum = 3 and we add one more element to it, and so on...

The calculation flow:

Or in the form of a table, where each row represents a function call on the next array element:

sum	current	result
-----	---------	--------

the first call		
----------------	--	--

0		
---	--	--

1		
---	--	--

1		
---	--	--

the second call		
-----------------	--	--

1	2	
---	---	--

2		
---	--	--

3		
---	--	--

the third call

3
3
6

the fourth call

6
4
10

the fifth call

10
5
15

Here we can clearly see how the result of the previous call becomes the first argument of the next one.

We also can omit the initial value:

```
let arr = [1, 2, 3, 4, 5];  
  
// removed initial value from reduce (no 0)  
let result = arr.reduce((sum, current) => sum + current);  
  
alert( result ); // 15
```

The result is the same. That's because if there's no initial, then reduce takes the first element of the array as the initial value and starts the iteration from the 2nd element.

The calculation table is the same as above, minus the first row.
But such use requires an extreme care. If the array is empty, then reduce call without initial value gives an error.
Here's an example:

```
let arr = [];  
  
// Error: Reduce of empty array with no initial value  
// if the initial value existed, reduce would return it for the empty arr.  
arr.reduce((sum, current) => sum + current);
```

So it's advised to always specify the initial value.
The method `arr.reduceRight` does the same, but goes from right to left.
`Array.isArray` does not form a separate language type. They are based on objects.
So `typeof` does not help to distinguish a plain object from an array:

```
alert(typeof {}); // object  
alert(typeof []); // object (same)
```

...But arrays are used so often that there's a special method for that:

`Array.isArray(value)`. It returns true if the value is an array, and false otherwise.

```
alert(Array.isArray({})); // false
```

```
alert(Array.isArray([])); // true
```

Most methods support “thisArg” Almost all array methods that call functions – like `find`, `filter`, `map`, with a notable exception of `sort`, accept an optional additional parameter `thisArg`.

That parameter is not explained in the sections above, because it’s rarely used. But for completeness we have to cover it.

Here’s the full syntax of these methods:

```
arr.find(func, thisArg);  
arr.filter(func, thisArg);  
arr.map(func, thisArg);  
// ...  
// thisArg is the optional last argument
```

The value of `thisArg` parameter becomes `this` for `func`.
For example, here we use a method of `array` object as a filter, and `thisArg` passes the context:

```

    let army = {
    minAge: 18,
    maxAge: 27,
    canJoin(user) {
      return user.age >= this.minAge && user.age < this.maxAge;
    }
  };

let users = [
  {age: 16},
  {age: 20},
  {age: 23},
  {age: 30}
];

// find users, for who army.canJoin returns true
let soldiers = users.filter(array.canJoin, array);

alert(soldiers.length); // 2
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23

```

If in the example above we used `users.filter(array.canJoin)`, then `array.canJoin` would be called as a standalone function, with `this=undefined`, thus leading to an instant error.

A call to `users.filter(array.canJoin, array)` can be replaced with `users.filter(user => array.canJoin(user))`, that does the same. The latter is used more often, as it's a bit easier to understand for most people.

SummaryA cheat sheet of array methods:

To add/remove elements:

`push(...items)` – adds items to the end,
`pop()` – extracts an item from the end,
`shift()` – extracts an item from the beginning,
`unshift(...items)` – adds items to the beginning.
`splice(pos, deleteCount, ...items)` – at index `pos` deletes `deleteCount` elements and

inserts items.

slice(start, end) – creates a new array, copies elements from index start till end (not inclusive) into it.

concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.

To search among elements:

indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.

includes(value) – returns true if the array has value, otherwise false.

find/filter(func) – filter elements through the function, return first/all values that make it return true.

findIndex is like find, but returns the index instead of a value.

To iterate over elements:

forEach(func) – calls func for every element, does not return anything.

To transform the array:

map(func) – creates a new array from results of calling func for every element.

sort(func) – sorts the array in-place, then returns it.

reverse() – reverses the array in-place, then returns it.

split/join – convert a string to array and back.

reduce/reduceRight(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

Additionally:

Array.isArray(value) checks value for being an array, if so returns true, otherwise false.

Please note that methods sort, reverse and splice modify the array itself.

These methods are the most used ones, they cover 99% of use cases. But there are few others:

`arr.some(fn)/arr.every(fn)` check the array.

The function `fn` is called on each element of the array similar to `map`. If any/all results are true, returns true, otherwise false.

These methods behave sort of like `||` and `&&` operators: if `fn` returns a truthy value, `arr.some()` immediately returns true and stops iterating over the rest of items; if `fn` returns a falsy value, `arr.every()` immediately returns false and stops iterating over the rest of items as well.

We can use `every` to compare arrays:

```
function arraysEqual(arr1, arr2) {  
  return arr1.length === arr2.length && arr1.every((value, index) => value ===  
    arr2[index]);  
}
```

```
alert( arraysEqual([1, 2], [1, 2])); // true
```

`arr.fill(value, start, end)` – fills the array with repeating value from index start to end.

`arr.copyWithin(target, start, end)` – copies its elements from position start till position end into itself, at position target (overwrites existing).

`arr.flat(depth)/arr.flatMap(fn)` create a new flat array from a multidimensional array.

For the full list, see the manual.

From the first sight it may seem that there are so many methods, quite difficult to remember. But actually that's much easier.

Look through the cheat sheet just to be aware of them. Then solve the tasks of this

chapter to practice, so that you have experience with array methods.

Afterwards whenever you need to do something with an array, and you don't know how – come here, look at the cheat sheet and find the right method. Examples will help you to write it correctly. Soon you'll automatically remember the methods, without specific efforts from your side.

Tasks Translate border-left-width to borderWidth importance: 5 Write the function camelize(str) that changes dash-separated words like “my-short-string” into camel-cased “myShortString”.

That is: removes all dashes, each word after dash becomes uppercased.

Examples:

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

P.S. Hint: use split to split the string into an array, transform it and join back.
Open a sandbox with tests. solution

```
function camelize(str) {
return str
.split('-') // splits 'my-long-word' into array ['my', 'long', 'word']
.map(
// capitalizes first letters of all array items except the first one
// converts ['my', 'long', 'word'] into ['my', 'Long', 'Word']
(word, index) => index == 0 ? word : word[0].toUpperCase() + word.slice(1)
)
.join(""); // joins ['my', 'Long', 'Word'] into 'myLongWord'
}
```

Open the solution with tests in a sandbox. Filter range importance: 4 Write a function filterRange(arr, a, b) that gets an array arr, looks for elements with values higher or equal to a and lower or equal to b and return a result as an array.

The function should not modify the array. It should return the new array.

For instance:

```
let arr = [5, 3, 8, 1];  
  
let filtered = filterRange(arr, 1, 4);  
  
alert( filtered ); // 3,1 (matching values)  
  
alert( arr ); // 5,3,8,1 (not modified)
```

Open a sandbox with tests.solution

```
function filterRange(arr, a, b) {  
  // added brackets around the expression for better readability  
  return arr.filter(item => (a <= item && item <= b));  
}  
  
let arr = [5, 3, 8, 1];  
  
let filtered = filterRange(arr, 1, 4);  
  
alert( filtered ); // 3,1 (matching values)  
  
alert( arr ); // 5,3,8,1 (not modified)
```

Open the solution with tests in a sandbox. Filter range "in place" importance: 4 Write a function `filterRangeInPlace(arr, a, b)` that gets an array `arr` and removes from it all values except those that are between `a` and `b`. The test is: `a "d arr[i] "d b`. The function should only modify the array. It should not return anything. For instance:

```
let arr = [5, 3, 8, 1];  
filterRangeInPlace(arr, 1, 4); // removed the numbers except from 1 to 4  
alert( arr ); // [3, 1]
```

Open a sandbox with tests.solution

```
function filterRangeInPlace(arr, a, b) {  
  for (let i = 0; i < arr.length; i++) {  
    let val = arr[i];  
  
    // remove if outside of the interval  
    if (val < a || val > b) {  
      arr.splice(i, 1);  
      i--;  
    }  
  }  
}  
  
let arr = [5, 3, 8, 1];  
filterRangeInPlace(arr, 1, 4); // removed the numbers except from 1 to 4  
alert( arr ); // [3, 1]
```

Open the solution with tests in a sandbox.Sort in decreasing orderimportance: 4

```
let arr = [5, 2, 1, -10, 8];  
  
// ... your code to sort it in decreasing order  
  
alert( arr ); // 8, 5, 2, 1, -10
```

solution

```
let arr = [5, 2, 1, -10, 8];  
  
arr.sort((a, b) => b - a);  
  
alert( arr );
```

Copy and sort array importance: 5 We have an array of strings `arr`. We'd like to have a sorted copy of it, but keep `arr` unmodified.
Create a function `copySorted(arr)` that returns such a copy.

```
let arr = ["HTML", "JavaScript", "CSS"];  
  
let sorted = copySorted(arr);  
  
alert( sorted ); // CSS, HTML, JavaScript  
alert( arr ); // HTML, JavaScript, CSS (no changes)
```

solution We can use `slice()` to make a copy and run the sort on it:


```
function copySorted(arr) {  
  return arr.slice().sort();  
}  
  
let arr = ["HTML", "JavaScript", "CSS"];  
  
let sorted = copySorted(arr);  
  
alert( sorted );  
alert( arr );
```

Create an extendable calculator importance: 5 Create a constructor function Calculator that creates “extendable” calculator objects. The task consists of two parts.

First, implement the method calculate(str) that takes a string like "1 + 2" in the format “NUMBER operator NUMBER” (space-delimited) and returns the result. Should understand plus + and minus -. Usage example:

```
let calc = new Calculator;  
  
alert( calc.calculate("3 + 7") ); // 10
```

Then add the method `addMethod(name, func)` that teaches the calculator a new operation. It takes the operator name and the two-argument function `func(a,b)` that implements it.

For instance, let's add the multiplication `*`, division `/` and power `**`:

```
let powerCalc = new Calculator;
powerCalc.addMethod("**", (a, b) => a * b);
powerCalc.addMethod("/", (a, b) => a / b);
powerCalc.addMethod("**", (a, b) => a ** b);
```

```
let result = powerCalc.calculate("2 ** 3");
alert( result ); // 8
```

No parentheses or complex expressions in this task.

The numbers and the operator are delimited with exactly one space.

There may be error handling if you'd like to add it.

Open a sandbox with tests.[solution](#)

Please note how methods are stored. They are simply added to this.methods property.

All tests and numeric conversions are done in the calculate method. In future it may be extended to support more complex expressions.

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
        a = +split[0],
        op = split[1],
```

```

    b = +split[2];

    if (!this.methods[op] || isNaN(a) || isNaN(b)) {
        return NaN;
    }

    return this.methods[op](a, b);
};

this.addMethod = function(name, func) {
    this.methods[name] = func;
};
}

```

Open the solution with tests in a sandbox. [Map to names](#) importance: 5 You have an array of user objects, each one has user.name. Write the code that converts it into an array of names.
For instance:

```

    let john = { name: "John", age: 25 };
    let pete = { name: "Pete", age: 30 };
    let mary = { name: "Mary", age: 28 };

    let users = [ john, pete, mary ];

    let names = /* ... your code */

    alert( names ); // John, Pete, Mary

```

solution

```

    let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(item => item.name);

alert( names ); // John, Pete, Mary

```

Map to objectsimportance: 5You have an array of user objects, each one has name, surname and id.
Write the code to create another array from it, of objects with id and fullName, where fullName is generated from name and surname.
For instance:

```

    let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = /* ... your code ... */

/*
usersMapped = [
  { fullName: "John Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ) // 1
alert( usersMapped[0].fullName ) // John Smith

```

So, actually you need to map one array of objects to another. Try using => here.
There's a small catch.

solution

```
    let john = { name: "John", surname: "Smith", id: 1 };
    let pete = { name: "Pete", surname: "Hunt", id: 2 };
    let mary = { name: "Mary", surname: "Key", id: 3 };

    let users = [ john, pete, mary ];

    let usersMapped = users.map(user => ({
      fullName: `${user.name} ${user.surname}`,
      id: user.id
    }));

    /*
    usersMapped = [
      { fullName: "John Smith", id: 1 },
      { fullName: "Pete Hunt", id: 2 },
      { fullName: "Mary Key", id: 3 }
    ]
    */

    alert( usersMapped[0].id ); // 1
    alert( usersMapped[0].fullName ); // John Smith
```

Please note that in the arrow functions we need to use additional brackets.
We can't write like this:

```
    let usersMapped = users.map(user => {
      fullName: `${user.name} ${user.surname}`,
      id: user.id
    });
```

As we remember, there are two arrow functions: without body value `=> expr` and with body value `=> {...}`. Here JavaScript would treat `{` as the start of function body, not the start of the object. The workaround is to wrap them in the “normal” brackets:

```
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

Now fine.

Sort users by ageimportance: 5Write the function `sortByAge(users)` that gets an array of objects with the age property and sorts them by age. For instance:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// now: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

solution

```

    function sortByAge(arr) {
    arr.sort((a, b) => a.age - b.age);
    }

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// now sorted is: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete

```

Shuffle an arrayimportance: 3Write the function `shuffle(array)` that shuffles (randomly reorders) elements of the array. Multiple runs of `shuffle` may lead to different orders of elements. For instance:

```

let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...

```

All element orders should have an equal probability. For instance, [1,2,3] can be reordered as [1,2,3] or [1,3,2] or [3,1,2] etc, with equal probability of each case.
The simple solution could be:

```
function shuffle(array) {  
  array.sort(() => Math.random() - 0.5);  
}
```

```
let arr = [1, 2, 3];  
shuffle(arr);  
alert(arr);
```

That somewhat works, because `Math.random() - 0.5` is a random number that may be positive or negative, so the sorting function reorders elements randomly. But because the sorting function is not meant to be used this way, not all permutations have the same probability. For instance, consider the code below. It runs `shuffle` 1000000 times and counts appearances of all possible results:

```
function shuffle(array) {  
  array.sort(() => Math.random() - 0.5);
```



```

}

// counts of appearances for all possible permutations
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join("")]++;
}

// show counts of all possible permutations
for (let key in count) {
  alert(` ${key}: ${count[key]} `);
}

```

An example result (depends on JS engine):

```

    123: 250706
132: 124425
213: 249618
231: 124880
312: 125148
321: 125223

```

We can see the bias clearly: 123 and 213 appear much more often than others. The result of the code may vary between JavaScript engines, but we can already see that the approach is unreliable.

Why it doesn't work? Generally speaking, sort is a "black box": we throw an array and a comparison function into it and expect the array to be sorted. But due to the utter randomness of the comparison the black box goes mad, and how exactly it goes mad depends on the concrete implementation that differs between engines.

There are other good ways to do the task. For instance, there's a great algorithm called Fisher-Yates shuffle. The idea is to walk the array in the reverse order and swap each element with a random one before it:

```
function shuffle(array) {  
  for (let i = array.length - 1; i > 0; i--) {  
    let j = Math.floor(Math.random() * (i + 1)); // random index from 0 to i  
  
    // swap elements array[i] and array[j]  
    // we use "destructuring assignment" syntax to achieve that  
    // you'll find more details about that syntax in later chapters  
    // same can be written as:  
    // let t = array[i]; array[i] = array[j]; array[j] = t  
    [array[i], array[j]] = [array[j], array[i]];  
  }  
}
```

Let's test it the same way:

```
function shuffle(array) {  
  for (let i = array.length - 1; i > 0; i--) {  
    let j = Math.floor(Math.random() * (i + 1));  
    [array[i], array[j]] = [array[j], array[i]];  
  }  
}  
  
// counts of appearances for all possible permutations  
let count = {  
  '123': 0,  
  '132': 0,
```

```

'213': 0,
'231': 0,
'321': 0,
'312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join("")]++;
}

// show counts of all possible permutations
for (let key in count) {
  alert(` ${key}: ${count[key]} `);
}

```

The example output:

```

      123: 166693
132: 166647
213: 166628
231: 167517
312: 166199
321: 166316

```

Looks good now: all permutations appear with the same probability. Also, performance-wise the Fisher-Yates algorithm is much better, there's no "sorting" overhead.

Get average ageimportance: 4Write the function `getAverageAge(users)` that gets an array of objects with property `age` and returns the average age.

The formula for the average is $(age1 + age2 + \dots + ageN) / N$.

For instance:

```

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };

```

```
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

solution

```
function getAverageAge(users) {
  return users.reduce((prev, user) => prev + user.age, 0) / users.length;
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // 28
```

Filter unique array members importance: 4 Let arr be an array.
Create a function unique(arr) that should return an array with unique items of arr.
For instance:

```
function unique(arr) {
  /* your code */
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
```

```
"Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(strings) ); // Hare, Krishna, :-O
```

Open a sandbox with tests.solutionLet's walk the array items:

For each item we'll check if the resulting array already has that item.
If it is so, then ignore, otherwise add to results.

```
function unique(arr) {
  let result = [];

  for (let str of arr) {
    if (!result.includes(str)) {
      result.push(str);
    }
  }

  return result;
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(strings) ); // Hare, Krishna, :-O
```

The code works, but there's a potential performance problem in it.
The method `result.includes(str)` internally walks the array `result` and compares each

element against str to find the match.

So if there are 100 elements in result and no one matches str, then it will walk the whole result and do exactly 100 comparisons. And if result is large, like 10000, then there would be 10000 comparisons.

That's not a problem by itself, because JavaScript engines are very fast, so walk 10000 array is a matter of microseconds.

But we do such test for each element of arr, in the for loop.

So if arr.length is 10000 we'll have something like $10000 \times 10000 = 100$ millions of comparisons. That's a lot.

So the solution is only good for small arrays.

Further in the chapter Map and Set we'll see how to optimize it.

Open the solution with tests in a sandbox. Create keyed object from array importance:

4Let's say we received an array of users in the form {id:..., name:..., age:... }.

Create a function groupById(arr) that creates an object from it, with id as the key, and array items as values.

For example:

```
let users = [
  {id: 'john', name: "John Smith", age: 20},
  {id: 'ann', name: "Ann Smith", age: 24},
  {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);

/*
// after the call we should have:

usersById = {
  john: {id: 'john', name: "John Smith", age: 20},
  ann: {id: 'ann', name: "Ann Smith", age: 24},
  pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
*/
```

Such function is really handy when working with server data.

In this task we assume that id is unique. There may be no two array items with the same id.

Please use array .reduce method in the solution.

Open a sandbox with tests.solution

```

    function groupById(array) {
    return array.reduce((obj, value) => {
        obj[value.id] = value;
        return obj;
    }, {})
    }

```

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/Array-methods","identifier":"Array-methods"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Data types","item":"https://javascript.info/data-types"]}]July 15, 2022IterablesIterable objects are a generalization of arrays. That's a concept that allows us to make any object useable in a for..of loop. Of course, Arrays are iterable. But there are many other built-in objects, that are iterable as well. For instance, strings are also iterable.

If an object isn't technically an array, but represents a collection (list, set) of something, then for..of is a great syntax to loop over it, so let's see how to make it work.

Symbol.iteratorWe can easily grasp the concept of iterables by making one of our own. For instance, we have an object that is not an array, but looks suitable for for..of. Like a range object that represents an interval of numbers:

```

    let range = {
    from: 1,
    to: 5
    };

    // We want the for..of to work:
    // for(let num of range) ... num=1,2,3,4,5

```

To make the range object iterable (and thus let for..of work) we need to add a method to the object named Symbol.iterator (a special built-in symbol just for that).

When for..of starts, it calls that method once (or errors if not found). The method must return an iterator – an object with the method next.

Onward, for..of works only with that returned object.

When for..of wants the next value, it calls next() on that object.

The result of next() must have the form {done: Boolean, value: any}, where done=true means that the loop is finished, otherwise value is the next value.

Here's the full implementation for range with remarks:

```
    let range = {
  from: 1,
  to: 5
};

// 1. call to for..of initially calls this
range[Symbol.iterator] = function() {

  // ...it returns the iterator object:
  // 2. Onward, for..of works only with the iterator object below, asking it for next values
  return {
    current: this.from,
    last: this.to,

    // 3. next() is called on each iteration by the for..of loop
    next() {
      // 4. it should return the value as an object {done:..., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
}
```



```
    }  
  }  
};  
};
```

```
// now it works!  
for (let num of range) {  
  alert(num); // 1, then 2, 3, 4, 5  
}
```

Please note the core feature of iterables: separation of concerns.

The range itself does not have the `next()` method.
Instead, another object, a so-called “iterator” is created by the call to `range[Symbol.iterator]()`, and its `next()` generates values for the iteration.

So, the iterator object is separate from the object it iterates over.
Technically, we may merge them and use range itself as the iterator to make the code simpler.
Like this:

```
let range = {  
  from: 1,  
  to: 5,  
  
  [Symbol.iterator]() {  
    this.current = this.from;  
    return this;  
  },  
  
  next() {  
    if (this.current <= this.to) {  
      return { done: false, value: this.current++ };  
    }  
  }  
};
```

```

    } else {
      return { done: true };
    }
  }
};

for (let num of range) {
  alert(num); // 1, then 2, 3, 4, 5
}

```

Now `range[Symbol.iterator]()` returns the range object itself: it has the necessary `next()` method and remembers the current iteration progress in `this.current`. Shorter? Yes. And sometimes that's fine too.

The downside is that now it's impossible to have two `for..of` loops running over the object simultaneously: they'll share the iteration state, because there's only one iterator – the object itself. But two parallel `for-ofs` is a rare thing, even in async scenarios.

Infinite iterators

Infinite iterators are also possible. For instance, the range becomes infinite for `range.to = Infinity`. Or we can make an iterable object that generates an infinite sequence of pseudorandom numbers. Also can be useful.

There are no limitations on `next`, it can return more and more values, that's normal. Of course, the `for..of` loop over such an iterable would be endless. But we can always stop it using `break`.

String is iterable Arrays and strings are most widely used built-in iterables. For a string, `for..of` loops over its characters:

```

    for (let char of "test") {
      // triggers 4 times: once for each character
      alert( char ); // t, then e, then s, then t
    }

```

And it works correctly with surrogate pairs!

```
let str = 'Ø5Ü³Ø=Þ ';
for (let char of str) {
  alert( char ); // Ø5Ü³, and then Ø=Þ
}
```

Calling an iterator explicitly For deeper understanding, let's see how to use an iterator explicitly. We'll iterate over a string in exactly the same way as `for..of`, but with direct calls. This code creates a string iterator and gets values from it "manually":

```
let str = "Hello";

// does the same as
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
```

```

let result = iterator.next();
if (result.done) break;
alert(result.value); // outputs characters one by one
}

```

That is rarely needed, but gives us more control over the process than `for..of`. For instance, we can split the iteration process: iterate a bit, then stop, do something else, and then resume later.

Iterables and array-likes Two official terms look similar, but are very different. Please make sure you understand them well to avoid the confusion.

Iterables are objects that implement the `Symbol.iterator` method, as described above. Array-likes are objects that have indexes and length, so they look like arrays.

When we use JavaScript for practical tasks in a browser or any other environment, we may meet objects that are iterables or array-likes, or both.

For instance, strings are both iterable (`for..of` works on them) and array-like (they have numeric indexes and length).

But an iterable may be not array-like. And vice versa an array-like may be not iterable.

For example, the range in the example above is iterable, but not array-like, because it does not have indexed properties and length.

And here's the object that is array-like, but not iterable:

```

let arrayLike = { // has indexes and length => array-like
  0: "Hello",
  1: "World",
  length: 2
};

// Error (no Symbol.iterator)
for (let item of arrayLike) {}

```

Both iterables and array-likes are usually not arrays, they don't have push, pop etc. That's rather inconvenient if we have such an object and want to work with it as with an array. E.g. we would like to work with range using array methods. How to achieve that? Array.from There's a universal method Array.from that takes an iterable or array-like value and makes a "real" Array from it. Then we can call array methods on it. For instance:

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (method works)
```

Array.from at the line (*) takes the object, examines it for being an iterable or array-like, then makes a new array and copies all items to it. The same happens for an iterable:

```
// assuming that range is taken from the example above
```

```
let arr = Array.from(range);  
alert(arr); // 1,2,3,4,5 (array toString conversion works)
```

The full syntax for `Array.from` also allows us to provide an optional “mapping” function:

```
Array.from(obj[, mapFn, thisArg])
```

The optional second argument `mapFn` can be a function that will be applied to each element before adding it to the array, and `thisArg` allows us to set this for it. For instance:

```
// assuming that range is taken from the example above  
  
// square each number  
let arr = Array.from(range, num => num * num);  
  
alert(arr); // 1,4,9,16,25
```

Here we use `Array.from` to turn a string into an array of characters:

```

let str = 'Ø5Ü³Ø=Þ ';

// splits str into array of characters
let chars = Array.from(str);

alert(chars[0]); // Ø5Ü³
alert(chars[1]); // Ø=Þ
alert(chars.length); // 2

```

Unlike `str.split`, it relies on the iterable nature of the string and so, just like `for..of`, correctly works with surrogate pairs. Technically here it does the same as:

```

let str = 'Ø5Ü³Ø=Þ ';

let chars = []; // Array.from internally does the same loop
for (let char of str) {
  chars.push(char);
}

alert(chars);

```

...But it is shorter.
We can even build surrogate-aware slice on it:

```

    function slice(str, start, end) {
    return Array.from(str).slice(start, end).join("");
    }

let str = 'Ø5Ü³Ø=Þ ØgÝö';

alert( slice(str, 1, 3) ); // Ø=Þ ØgÝö

// the native method does not support surrogate pairs
alert( str.slice(1, 3) ); // garbage (two pieces from different surrogate pairs)

```

Summary Objects that can be used in for..of are called iterable.

Technically, iterables must implement the method named Symbol.iterator.

The result of obj[Symbol.iterator]() is called an iterator. It handles further iteration process.

An iterator must have the method named next() that returns an object {done: Boolean, value: any}, here done:true denotes the end of the iteration process, otherwise the value is the next value.

The Symbol.iterator method is called automatically by for..of, but we also can do it directly.

Built-in iterables like strings or arrays, also implement Symbol.iterator.

String iterator knows about surrogate pairs.

Objects that have indexed properties and length are called array-like. Such objects may also have other properties and methods, but lack the built-in methods of arrays.

If we look inside the specification – we'll see that most built-in methods assume that they work with iterables or array-likes instead of “real” arrays, because that's more abstract.

Array.from(obj[, mapFn, thisArg]) makes a real Array from an iterable or array-like obj,

and we can then use array methods on it. The optional arguments `mapFn` and `thisArg` allow us to apply a function to each item.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/iterable","identifier":"V iterable"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialThe JavaScript languageData types{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https:// javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Data types","item":"https:// javascript.info/data-types"}]}November 14, 2022Map and SetTill now, we've learned about the following complex data structures:

Objects are used for storing keyed collections.

Arrays are used for storing ordered collections.

But that's not enough for real life. That's why Map and Set also exist.

MapMap is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

Methods and properties are:

`new Map()` – creates the map.

`map.set(key, value)` – stores the value by the key.

`map.get(key)` – returns the value by the key, undefined if key doesn't exist in map.

`map.has(key)` – returns true if the key exists, false otherwise.

`map.delete(key)` – removes the element (the key/value pair) by the key.

`map.clear()` – removes everything from the map.

`map.size` – returns the current element count.

For instance:

```
let map = new Map();

map.set('1', 'str1'); // a string key
map.set(1, 'num1');   // a numeric key
map.set(true, 'bool1'); // a boolean key

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

As we can see, unlike objects, keys are not converted to strings. Any type of key is possible.

map[key] isn't the right way to use a Map
Although map[key] also works, e.g. we can set map[key] = 2, this is treating map as a plain JavaScript object, so it implies all corresponding limitations (only string/symbol keys and so on).
So we should use map methods: set, get and so on.

Map can also use objects as keys.
For instance:

```
let john = { name: "John" };

// for every user, let's store their visits count
let visitsCountMap = new Map();

// john is the key for the map
visitsCountMap.set(john, 123);
```

```
alert( visitsCountMap.get(john) ); // 123
```

Using objects as keys is one of the most notable and important Map features. The same does not count for Object. String as a key in Object is fine, but we can't use another Object as a key in Object. Let's try:

```
let john = { name: "John" };
let ben = { name: "Ben" };

let visitsCountObj = {}; // try to use an object

visitsCountObj[ben] = 234; // try to use ben object as the key
visitsCountObj[john] = 123; // try to use john object as the key, ben object will get
replaced

// That's what got written!
alert( visitsCountObj["[object Object]"] ); // 123
```

As visitsCountObj is an object, it converts all Object keys, such as john and ben above, to same string "[object Object]". Definitely not what we want.

How Map compares keys

To test keys for equivalence, Map uses the algorithm SameValueZero. It is roughly the same as strict equality ===, but the difference is that NaN is considered equal to NaN. So NaN can be used as the key as well. This algorithm can't be changed or customized.

Chaining

Every `map.set` call returns the map itself, so we can “chain” the calls:

```
map.set('1', 'str1')  
.set(1, 'num1')  
.set(true, 'bool1');
```

Iteration over Map For looping over a map, there are 3 methods:

`map.keys()` – returns an iterable for keys,
`map.values()` – returns an iterable for values,
`map.entries()` – returns an iterable for entries [key, value], it’s used by default in `for..of`.

For instance:

```
let recipeMap = new Map([  
  ['cucumber', 500],  
  ['tomatoes', 350],  
  ['onion', 50]  
]);  
  
// iterate over keys (vegetables)  
for (let vegetable of recipeMap.keys()) {  
  alert(vegetable); // cucumber, tomatoes, onion  
}  
  
// iterate over values (amounts)  
for (let amount of recipeMap.values()) {  
  alert(amount); // 500, 350, 50  
}
```

```
// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of recipeMap.entries()
  alert(entry); // cucumber,500 (and so on)
}
```

The insertion order is used

The iteration goes in the same order as the values were inserted. Map preserves this order, unlike a regular Object.

Besides that, Map has a built-in forEach method, similar to Array:

```
// runs the function for each (key, value) pair
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

Object.entries: Map from Object When a Map is created, we can pass an array (or another iterable) with key/value pairs for initialization, like this:

```
// array of [key, value] pairs
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);
```

```
alert( map.get('1') ); // str1
```

If we have a plain object, and we'd like to create a Map from it, then we can use built-in method `Object.entries(obj)` that returns an array of key/value pairs for an object exactly in that format.

So we can create a map from an object like this:

```
let obj = {  
  name: "John",  
  age: 30  
};
```

```
let map = new Map(Object.entries(obj));
```

```
alert( map.get('name') ); // John
```

Here, `Object.entries` returns the array of key/value pairs: `[["name","John"], ["age", 30]]`. That's what Map needs.

`Object.fromEntries`: Object from Map We've just seen how to create Map from a plain object with `Object.entries(obj)`.

There's `Object.fromEntries` method that does the reverse: given an array of `[key, value]` pairs, it creates an object from them:

```
    let prices = Object.fromEntries([
    ['banana', 1],
    ['orange', 2],
    ['meat', 4]
    ]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

We can use `Object.fromEntries` to get a plain object from `Map`.
E.g. we store the data in a `Map`, but we need to pass it to a 3rd-party code that expects a plain object.
Here we go:

```
    let map = new Map();
    map.set('banana', 1);
    map.set('orange', 2);
    map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // make a plain object (*)

// done!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

A call to `map.entries()` returns an iterable of key/value pairs, exactly in the right

format for `Object.fromEntries`.
We could also make line (*) shorter:

```
let obj = Object.fromEntries(map); // omit .entries()
```

That's the same, because `Object.fromEntries` expects an iterable object as the argument. Not necessarily an array. And the standard iteration for `map` returns same key/value pairs as `map.entries()`. So we get a plain object with same key/values as the `map`.

`Set` is a special type collection – “set of values” (without keys), where each value may occur only once.

Its main methods are:

`new Set([iterable])` – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.

`set.add(value)` – adds a value, returns the set itself.

`set.delete(value)` – removes the value, returns true if value existed at the moment of the call, otherwise false.

`set.has(value)` – returns true if the value exists in the set, otherwise false.

`set.clear()` – removes everything from the set.

`set.size` – is the elements count.

The main feature is that repeated calls of `set.add(value)` with the same value don't do anything. That's the reason why each value appears in a `Set` only once.

For example, we have visitors coming, and we'd like to remember everyone. But repeated visits should not lead to duplicates. A visitor must be “counted” only once.

`Set` is just the right thing for that:

```
let set = new Set();
```



```

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set keeps only unique values
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // John (then Pete and Mary)
}

```

The alternative to Set could be an array of users, and the code to check for duplicates on every insertion using `arr.find`. But the performance would be much worse, because this method walks through the whole array checking every element. Set is much better optimized internally for uniqueness checks.

Iteration over Set We can loop over a set either with `for..of` or using `forEach`:

```

let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// the same with forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});

```

Note the funny thing. The callback function passed in `forEach` has 3 arguments: a value, then the same value `valueAgain`, and then the target object. Indeed, the same value appears in the arguments twice.

That's for compatibility with `Map` where the callback passed `forEach` has three arguments. Looks a bit strange, for sure. But this may help to replace `Map` with `Set` in certain cases with ease, and vice versa.

The same methods `Map` has for iterators are also supported:

`set.keys()` – returns an iterable object for values,

`set.values()` – same as `set.keys()`, for compatibility with `Map`,

`set.entries()` – returns an iterable object for entries `[value, value]`, exists for compatibility with `Map`.

`SummaryMap` – is a collection of keyed values.

Methods and properties:

`new Map([iterable])` – creates the map, with optional iterable (e.g. array) of `[key,value]` pairs for initialization.

`map.set(key, value)` – stores the value by the key, returns the map itself.

`map.get(key)` – returns the value by the key, undefined if key doesn't exist in map.

`map.has(key)` – returns true if the key exists, false otherwise.

`map.delete(key)` – removes the element by the key, returns true if key existed at the moment of the call, otherwise false.

`map.clear()` – removes everything from the map.

`map.size` – returns the current element count.

The differences from a regular `Object`:

Any keys, objects can be keys.

Additional convenient methods, the size property.

`Set` – is a collection of unique values.

Methods and properties:

`new Set([iterable])` – creates the set, with optional iterable (e.g. array) of values for initialization.

`set.add(value)` – adds a value (does nothing if value exists), returns the set itself.

`set.delete(value)` – removes the value, returns true if value existed at the moment of the call, otherwise false.

`set.has(value)` – returns true if the value exists in the set, otherwise false.

`set.clear()` – removes everything from the set.

`set.size` – is the elements count.

Iteration over `Map` and `Set` is always in the insertion order, so we can't say that these

collections are unordered, but we can't reorder elements or directly get an element by its number.

TasksFilter unique array membersimportance: 5Let arr be an array.

Create a function unique(arr) that should return an array with unique items of arr.

For instance:

```
function unique(arr) {  
  /* your code */  
}  
  
let values = ["Hare", "Krishna", "Hare", "Krishna",  
  "Krishna", "Krishna", "Hare", "Hare", ":-O"  
];  
  
alert( unique(values) ); // Hare, Krishna, :-O
```

P.S. Here strings are used, but can be values of any type.

P.P.S. Use Set to store unique values.

Open a sandbox with tests.solution

```
function unique(arr) {  
  return Array.from(new Set(arr));  
}
```

Open the solution with tests in a sandbox.Filter anagramsimportance: 4Anagrams are words that have the same number of same letters, but in different order.

For instance:

nap - pan
ear - are - era
cheaters - hectares - teachers

Write a function `aclean(arr)` that returns an array cleaned from anagrams.
For instance:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];  
alert( aclean(arr) ); // "nap,teachers,ear" or "PAN,cheaters,era"
```

From every anagram group should remain only one word, no matter which one.
Open a sandbox with tests.
To find all anagrams, let's split every word to letters and sort them. When letter-sorted, all anagrams are same.
For instance:

```
nap, pan -> anp  
ear, era, are -> aer  
cheaters, hectares, teachers -> aceehrst  
...
```

We'll use the letter-sorted variants as map keys to store only one value per each key:

```
function aclean(arr) {  
  let map = new Map();  
  
  for (let word of arr) {  
    // split the word by letters, sort them and join back
```

```

    let sorted = word.toLowerCase().split('').sort().join(''); // (*)
    map.set(sorted, word);
  }

  return Array.from(map.values());
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert( aclean(arr) );

```

Letter-sorting is done by the chain of calls in the line (*).
For convenience let's split it into multiple lines:

```

    let sorted = word // PAN
    .toLowerCase() // pan
    .split('') // ['p','a','n']
    .sort() // ['a','n','p']
    .join(''); // anp

```

Two different words 'PAN' and 'nap' receive the same letter-sorted form 'anp'.
The next line put the word into the map:

```

    map.set(sorted, word);

```

If we ever meet a word the same letter-sorted form again, then it would overwrite the previous value with the same key in the map. So we'll always have at maximum one word per letter-form.

At the end `Array.from(map.values())` takes an iterable over map values (we don't need keys in the result) and returns an array of them.

Here we could also use a plain object instead of the Map, because keys are strings.
That's how the solution can look:

```

    function aclean(arr) {
    let obj = {};

    for (let i = 0; i < arr.length; i++) {
        let sorted = arr[i].toLowerCase().split("").sort().join("");
        obj[sorted] = arr[i];
    }

    return Object.values(obj);
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert( aclean(arr) );

```

Open the solution with tests in a sandbox. ⁵importance: 5 We'd like to get an array of `map.keys()` in a variable and then apply array-specific methods to it, e.g. `.push`. But that doesn't work:

```

let map = new Map();

map.set("name", "John");

```

```
let keys = map.keys();

// Error: keys.push is not a function
keys.push("more");
```

Why? How can we fix the code to make keys.push work?
solutionThat's because map.keys() returns an iterable, but not an array.
We can convert it into an array using Array.from:

```
let map = new Map();

map.set("name", "John");

let keys = Array.from(map.keys());

keys.push("more");

alert(keys); // name, more
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this
before commenting...If you have suggestions what to improve - please submit a GitHub
issue or a pull request instead of commenting.If you can't understand something in the
article – please elaborate.To insert few words of code, use the <code> tag, for several
lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,
codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://javascript.info/map-set","identifier":"Vmap-
set"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageData types{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}],

```
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"},{"@type":"ListItem","position":3,"name":"Data types","item":"https://  
javascript.info/data-types"]}]}
```

November 13, 2022 WeakMap and WeakSet
As we know from the chapter Garbage collection, JavaScript engine keeps a value in memory while it is “reachable” and can potentially be used.

For instance:

```
let john = { name: "John" };

// the object can be accessed, john is the reference to it

// overwrite the reference
john = null;

// the object will be removed from memory
```

Usually, properties of an object or elements of an array or another data structure are considered reachable and kept in memory while that data structure is in memory. For instance, if we put an object into an array, then while the array is alive, the object will be alive as well, even if there are no other references to it. Like this:

```
let john = { name: "John" };

let array = [ john ];

john = null; // overwrite the reference

// the object previously referenced by john is stored inside the array
// therefore it won't be garbage-collected
// we can get it as array[0]
```

Similar to that, if we use an object as the key in a regular Map, then while the Map exists, that object exists as well. It occupies memory and may not be garbage collected. For instance:


```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // overwrite the reference

// john is stored inside the map,
// we can get it by using map.keys()
```

WeakMap is fundamentally different in this aspect. It doesn't prevent garbage-collection of key objects.
Let's see what it means on examples.
WeakMap The first difference between Map and WeakMap is that keys must be objects, not primitive values:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // works fine (object key)

// can't use a string as the key
weakMap.set("test", "Whoops"); // Error, because "test" is not an object
```

Now, if we use an object as the key in it, and there are no other references to that object – it will be removed from memory (and from the map) automatically.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // overwrite the reference

// john is removed from memory!
```

Compare it with the regular Map example above. Now if john only exists as the key of WeakMap – it will be automatically deleted from the map (and memory). WeakMap does not support iteration and methods keys(), values(), entries(), so there's no way to get all keys or values from it. WeakMap has only the following methods:

```
weakMap.set(key, value)
weakMap.get(key)
weakMap.delete(key)
weakMap.has(key)
```

Why such a limitation? That's for technical reasons. If an object has lost all other references (like john in the code above), then it is to be garbage-collected automatically. But technically it's not exactly specified when the cleanup happens.

The JavaScript engine decides that. It may choose to perform the memory cleanup immediately or to wait and do the cleaning later when more deletions happen. So, technically, the current element count of a WeakMap is not known. The engine may have cleaned it up or not, or did it partially. For that reason, methods that access all keys/values are not supported.

Now, where do we need such a data structure?

Use case: additional data The main area of application for WeakMap is an additional data storage.

If we're working with an object that “belongs” to another code, maybe even a third-party library, and would like to store some data associated with it, that should only exist while the object is alive – then WeakMap is exactly what's needed.

We put the data to a WeakMap, using the object as the key, and when the object is garbage collected, that data will automatically disappear as well.

```
weakMap.set(john, "secret documents");
```

```
// if john dies, secret documents will be destroyed automatically
```

Let's look at an example.

For instance, we have code that keeps a visit count for users. The information is stored in a map: a user object is the key and the visit count is the value. When a user leaves (its object gets garbage collected), we don't want to store their visit count anymore. Here's an example of a counting function with Map:

```
// Ø=ÜÁ visitsCount.js
let visitsCountMap = new Map(); // map: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

And here's another part of the code, maybe another file using it:

```
// Ø=ÜÁ main.js
let john = { name: "John" };

countUser(john); // count his visits

// later john leaves us
john = null;
```

Now, john object should be garbage collected, but remains in memory, as it's a key in visitsCountMap. We need to clean visitsCountMap when we remove users, otherwise it will grow in memory indefinitely. Such cleaning can become a tedious task in complex architectures. We can avoid it by switching to WeakMap instead:

```

// Ø=ÜÁ visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}

```

Now we don't have to clean visitsCountMap. After john object becomes unreachable, by all means except as a key of WeakMap, it gets removed from memory, along with the information by that key from WeakMap.

Use case: caching Another common example is caching. We can store ("cache") results from a function, so that future calls on the same object can reuse it.

To achieve that, we can use Map (not optimal scenario):

```

// Ø=ÜÁ cache.js
let cache = new Map();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculations of the result for */ obj;

    cache.set(obj, result);
    return result;
  }

  return cache.get(obj);
}

```

```
// Now we use process() in another file:

// Ø=ÜÁ main.js
let obj = { /* let's say we have an object */ };

let result1 = process(obj); // calculated

// ...later, from another place of the code...
let result2 = process(obj); // remembered result taken from cache

// ...later, when the object is not needed any more:
obj = null;

alert(cache.size); // 1 (Ouch! The object is still in cache, taking memory!)
```

For multiple calls of `process(obj)` with the same object, it only calculates the result the first time, and then just takes it from cache. The downside is that we need to clean cache when the object is not needed any more.

If we replace `Map` with `WeakMap`, then this problem disappears. The cached result will be removed from memory automatically after the object gets garbage collected.

```
// Ø=ÜÁ cache.js
let cache = new WeakMap();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculate the result for */ obj;

    cache.set(obj, result);
    return result;
  }
}
```

```

    return cache.get(obj);
}

// Ø=ÜÁ main.js
let obj = { /* some object */ };

let result1 = process(obj);
let result2 = process(obj);

// ...later, when the object is not needed any more:
obj = null;

// Can't get cache.size, as it's a WeakMap,
// but it's 0 or soon be 0
// When obj gets garbage collected, cached data will be removed as well

```

WeakSetWeakSet behaves similarly:

It is analogous to Set, but we may only add objects to WeakSet (not primitives).
 An object exists in the set while it is reachable from somewhere else.
 Like Set, it supports add, has and delete, but not size, keys() and no iterations.

Being “weak”, it also serves as additional storage. But not for arbitrary data, rather for “yes/no” facts. A membership in WeakSet may mean something about the object.
 For instance, we can add users to WeakSet to keep track of those who visited our site:

```

let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John visited us

```

```

visitedSet.add(pete); // Then Pete
visitedSet.add(john); // John again

// visitedSet has 2 users now

// check if John visited?
alert(visitedSet.has(john)); // true

// check if Mary visited?
alert(visitedSet.has(mary)); // false

john = null;

// visitedSet will be cleaned automatically

```

The most notable limitation of WeakMap and WeakSet is the absence of iterations, and the inability to get all current content. That may appear inconvenient, but does not prevent WeakMap/WeakSet from doing their main job – be an “additional” storage of data for objects which are stored/managed at another place.

SummaryWeakMap is Map-like collection that allows only objects as keys and removes them together with associated value once they become inaccessible by other means. WeakSet is Set-like collection that stores only objects and removes them once they become inaccessible by other means.

Their main advantages are that they have weak reference to objects, so they can easily be removed by garbage collector.

That comes at the cost of not having support for clear, size, keys, values...

WeakMap and WeakSet are used as “secondary” data structures in addition to the “primary” object storage. Once the object is removed from the primary storage, if it is only found as the key of WeakMap or in a WeakSet, it will be cleaned up automatically.

TasksStore "unread" flagsimportance: 5There’s an array of messages:

```

let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

```

Your code can access it, but the messages are managed by someone else’s code. New messages are added, old ones are removed regularly by that code, and you don’t

know the exact moments when it happens.

Now, which data structure could you use to store information about whether the message “has been read”? The structure must be well-suited to give the answer “was it read?” for the given message object.

P.S. When a message is removed from messages, it should disappear from your structure as well.

P.P.S. We shouldn't modify message objects, add our properties to them. As they are managed by someone else's code, that may lead to bad consequences.

solutionLet's store read messages in WeakSet:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

let readMessages = new WeakSet();

// two messages have been read
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// readMessages has 2 elements

// ...let's read the first message again!
readMessages.add(messages[0]);
// readMessages still has 2 unique elements

// answer: was the message[0] read?
alert("Read message 0: " + readMessages.has(messages[0])); // true

messages.shift();
// now readMessages has 1 element (technically memory may be cleaned later)
```


The WeakSet allows to store a set of messages and easily check for the existence of a message in it.

It cleans up itself automatically. The tradeoff is that we can't iterate over it, can't get "all read messages" from it directly. But we can do it by iterating over all messages and filtering those that are in the set.

Another, different solution could be to add a property like message.isRead=true to a message after it's read. As messages objects are managed by another code, that's generally discouraged, but we can use a symbolic property to avoid conflicts.

Like this:

```
// the symbolic property is only known to our code
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Now third-party code probably won't see our extra property.

Although symbols allow to lower the probability of problems, using WeakSet is better from the architectural point of view.

Store read datesimportance: 5There's an array of messages as in the previous task. The situation is similar.

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

The question now is: which data structure you'd suggest to store the information: "when the message was read?".

In the previous task we only needed to store the "yes/no" fact. Now we need to store the date, and it should only remain in memory until the message is garbage collected.

P.S. Dates can be stored as objects of built-in Date class, that we'll cover later.

solutionTo store a date, we can use WeakMap:

```

    let messages = [
      {text: "Hello", from: "John"},
      {text: "How goes?", from: "John"},
      {text: "See you soon", from: "Alice"}
    ];

let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// Date object we'll study later

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/weakmap-weakset", "identifier": "Vweakmap-weakset" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageData types{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Data types", "item": "https://javascript.info/data-types"}]}

June 27, 2021Object.keys, values, entriesLet's step away from the individual data structures and talk about the iterations over them. In the previous chapter we saw methods map.keys(), map.values(), map.entries(). These methods are generic, there is a common agreement to use them for data structures. If we ever create a data structure of our own, we should implement them too. They are supported for:

Map
Set
Array

Plain objects also support similar methods, but the syntax is a bit different. Object.keys, values, entriesFor plain objects, the following methods are available:

Object.keys(obj) – returns an array of keys.
Object.values(obj) – returns an array of values.
Object.entries(obj) – returns an array of [key, value] pairs.

Please note the distinctions (compared to map for example):

Map
Object

Call syntax
map.keys()
Object.keys(obj), but not obj.keys()

Returns
iterable
“real” Array

The first difference is that we have to call `Object.keys(obj)`, and not `obj.keys()`. Why so? The main reason is flexibility. Remember, objects are a base of all complex structures in JavaScript. So we may have an object of our own like `data` that implements its own `data.values()` method. And we still can call `Object.values(data)` on it. The second difference is that `Object.*` methods return “real” array objects, not just an iterable. That’s mainly for historical reasons. For instance:

```
let user = {  
  name: "John",  
  age: 30  
};
```

```
Object.keys(user) = ["name", "age"]  
Object.values(user) = ["John", 30]  
Object.entries(user) = [ ["name", "John"], ["age", 30] ]
```

Here’s an example of using `Object.values` to loop over property values:

```

    let user = {
      name: "John",
      age: 30
    };

    // loop over values
    for (let value of Object.values(user)) {
      alert(value); // John, then 30
    }

```

Object.keys/values/entries ignore symbolic properties
 Just like a for..in loop, these methods ignore properties that use Symbol(...) as keys.

Usually that's convenient. But if we want symbolic keys too, then there's a separate method Object.getPrototypeOfSymbols that returns an array of only symbolic keys. Also, there exist a method Reflect.ownKeys(obj) that returns all keys.

Transforming objects
 Objects lack many methods that exist for arrays, e.g. map, filter and others.

If we'd like to apply them, then we can use Object.entries followed by Object.fromEntries:

Use Object.entries(obj) to get an array of key/value pairs from obj.
 Use array methods on that array, e.g. map, to transform these key/value pairs.
 Use Object.fromEntries(array) on the resulting array to turn it back into an object.

For example, we have an object with prices, and would like to double them:

```

    let prices = {
    banana: 1,
    orange: 2,
    meat: 4,
};

let doublePrices = Object.fromEntries(
  // convert prices to array, map each key/value pair into another pair
  // and then fromEntries gives back the object
  Object.entries(prices).map(entry => [entry[0], entry[1] * 2])
);

alert(doublePrices.meat); // 8

```

It may look difficult at first sight, but becomes easy to understand after you use it once or twice. We can make powerful chains of transforms this way.

TasksSum the propertiesimportance: 5There is a salaries object with arbitrary number of salaries.

Write the function sumSalaries(salaries) that returns the sum of all salaries using Object.values and the for..of loop.

If salaries is empty, then the result must be 0.

For instance:

```

    let salaries = {
    "John": 100,
    "Pete": 300,
    "Mary": 250
};

alert( sumSalaries(salaries) ); // 650

```

Open a sandbox with tests.solution

```

function sumSalaries(salaries) {

  let sum = 0;
  for (let salary of Object.values(salaries)) {
    sum += salary;
  }

  return sum; // 650
}

let salaries = {
  "John": 100,
  "Pete": 300,
  "Mary": 250
};

alert( sumSalaries(salaries) ); // 650

```

Or, optionally, we could also get the sum using `Object.values` and `reduce`:

```

// reduce loops over array of salaries,
// adding them up
// and returns the result
function sumSalaries(salaries) {
  return Object.values(salaries).reduce((a, b) => a + b, 0) // 650
}

```

Open the solution with tests in a [sandbox](#). Count properties importance: 5 Write a

function count(obj) that returns the number of properties in the object:

```
    let user = {
      name: 'John',
      age: 30
    };

    alert( count(user) ); // 2
```

Try to make the code as short as possible.
P.S. Ignore symbolic properties, count only “regular” ones.
Open a sandbox with tests.[solution](#)

```
function count(obj) {
  return Object.keys(obj).length;
}
```

Open the solution with tests in a sandbox.[Ctrl + !](#)•[Previous lesson](#)[Ctrl + !](#)[Next lesson](#)[Share](#)[Tutorial](#) [map](#) [Comments](#) [read this before commenting...](#) If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) `var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {url:"https://javascript.info/keys-values-entries", identifier:"\keys-values-entries"}); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true; TutorialThe JavaScript languageData types{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement": [{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"}, {"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"}, {"@type":"ListItem", "position":3, "name":"Data types", "item":"https://javascript.info/data-types"}]}April 14, 2022Destructuring assignmentThe two most used data structures in JavaScript are Object and Array.`

Objects allow us to create a single entity that stores data items by key.
Arrays allow us to gather data items into an ordered list.

Although, when we pass those to a function, it may need not be an object/array as a whole. It may need individual pieces.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.

Destructuring also works great with complex functions that have a lot of parameters, default values, and so on. Soon we’ll see that.

Array destructuringHere’s an example of how an array is destructured into variables:

```
// we have an array with the name and surname
let arr = ["John", "Smith"]

// destructuring assignment
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // John
alert(surname); // Smith
```

Now we can work with variables instead of array members.
It looks great when combined with split or other array-returning methods:

```
let [firstName, surname] = "John Smith".split(' ');
alert(firstName); // John
alert(surname); // Smith
```

As you can see, the syntax is simple. There are several peculiar details though.
Let’s see more examples, to better understand it.

“Destructuring” does not mean “destructive”.

It's called “destructuring assignment,” because it “deconstructurizes” by copying items into variables. But the array itself is not modified.

It's just a shorter way to write:

```
// let [firstName, surname] = arr;  
let firstName = arr[0];  
let surname = arr[1];
```

Ignore elements using commas

Unwanted elements of the array can also be thrown away via an extra comma:

```
// second element is not needed  
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
  
alert( title ); // Consul
```

In the code above, the second element of the array is skipped, the third one is assigned to title, and the rest of the array items is also skipped (as there are no variables for them).

Works with any iterable on the right-side

...Actually, we can use it with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]  
let [one, two, three] = new Set([1, 2, 3]);
```

That works, because internally a destructuring assignment works by iterating over the right value. It's a kind of syntax sugar for calling `for..of` over the value to the right of `=` and assigning the values.

Assign to anything at the left-side

We can use any “assignables” on the left side.

For instance, an object property:

```
let user = {};  
[user.name, user.surname] = "John Smith".split(' ');  
  
alert(user.name); // John  
alert(user.surname); // Smith
```

Looping with `.entries()`

In the previous chapter we saw the `Object.entries(obj)` method.

We can use it with destructuring to loop over keys-and-values of an object:

```
    let user = {  
    name: "John",  
    age: 30  
};  
  
// loop over keys-and-values  
for (let [key, value] of Object.entries(user)) {  
    alert( `${key}:${value}` ); // name:John, then age:30  
}
```

The similar code for a Map is simpler, as it's iterable:

```
    let user = new Map();  
    user.set("name", "John");  
    user.set("age", "30");  
  
// Map iterates as [key, value] pairs, very convenient for destructuring  
for (let [key, value] of user) {  
    alert( `${key}:${value}` ); // name:John, then age:30  
}
```

Swap variables trick

There's a well-known trick for swapping values of two variables using a destructuring assignment:

```
let guest = "Jane";  
let admin = "Pete";  
  
// Let's swap the values: make guest=Pete, admin=Jane  
[guest, admin] = [admin, guest];  
  
alert(`${guest} ${admin}`); // Pete Jane (successfully swapped!)
```

Here we create a temporary array of two variables and immediately destructure it in swapped order.

We can swap more than two variables this way.

The rest '...' Usually, if the array is longer than the list at the left, the “extra” items are omitted.

For example, here only two items are taken, and the rest is just ignored:

```
let [name1, name2] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
  
alert(name1); // Julius
```

```
alert(name2); // Caesar  
// Further items aren't assigned anywhere
```

If we'd like also to gather all that follows – we can add one more parameter that gets “the rest” using three dots “...”:

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman  
Republic"];  
  
// rest is array of items, starting from the 3rd one  
alert(rest[0]); // Consul  
alert(rest[1]); // of the Roman Republic  
alert(rest.length); // 2
```

The value of rest is the array of the remaining array elements.
We can use any other variable name in place of rest, just make sure it has three dots before it and goes last in the destructuring assignment.

```
let [name1, name2, ...titles] = ["Julius", "Caesar", "Consul", "of the Roman  
Republic"];
```

```
// now titles = ["Consul", "of the Roman Republic"]
```

Default values If the array is shorter than the list of variables at the left, there'll be no errors. Absent values are considered undefined:

```
let [firstName, surname] = [];  
  
alert(firstName); // undefined  
alert(surname); // undefined
```

If we want a “default” value to replace the missing one, we can provide it using =:

```
// default values  
let [name = "Guest", surname = "Anonymous"] = ["Julius"];  
  
alert(name); // Julius (from array)  
alert(surname); // Anonymous (default used)
```

Default values can be more complex expressions or even function calls. They are evaluated only if the value is not provided.
For instance, here we use the prompt function for two defaults:

```
// runs only prompt for surname
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name); // Julius (from array)
alert(surname); // whatever prompt gets
```

Please note: the prompt will run only for the missing value (surname).
Object destructuringThe destructuring assignment also works with objects.
The basic syntax is:

```
let {var1, var2} = {var1:..., var2:...}
```

We should have an existing object on the right side, that we want to split into variables. The left side contains an object-like “pattern” for corresponding properties. In the simplest case, that’s a list of variable names in {...}.
For instance:

```
    let options = {  
      title: "Menu",  
      width: 100,  
      height: 200  
    };
```

```
let {title, width, height} = options;
```

```
alert(title); // Menu  
alert(width); // 100  
alert(height); // 200
```

Properties `options.title`, `options.width` and `options.height` are assigned to the corresponding variables.

The order does not matter. This works too:

```
    // changed the order in let {...}  
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

The pattern on the left side may be more complex and specify the mapping between properties and variables.

If we want to assign a property to a variable with another name, for instance, make `options.width` go into the variable named `w`, then we can set the variable name using a colon:


```

    let options = {
    title: "Menu",
    width: 100,
    height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w);     // 100
alert(h);     // 200

```

The colon shows “what : goes where”. In the example above the property width goes to w, property height goes to h, and title is assigned to the same name. For potentially missing properties we can set default values using “=”, like this:

```

    let options = {
    title: "Menu"
};

let {width = 100, height = 200, title} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200

```

Just like with arrays or function parameters, default values can be any expressions or even function calls. They will be evaluated if the value is not provided. In the code below prompt asks for width, but not for title:

```
let options = {  
  title: "Menu"  
};  
  
let {width = prompt("width?"), title = prompt("title?")} = options;  
  
alert(title); // Menu  
alert(width); // (whatever the result of prompt is)
```

We also can combine both the colon and equality:

```
let options = {  
  title: "Menu"  
};  
  
let {width: w = 100, height: h = 200, title} = options;
```

```
alert(title); // Menu
alert(w);     // 100
alert(h);     // 200
```

If we have a complex object with many properties, we can extract only what we need:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// only extract title as a variable
let { title } = options;

alert(title); // Menu
```

The rest pattern “...”What if the object has more properties than we have variables? Can we take some and then assign the “rest” somewhere? We can use the rest pattern, just like we did with arrays. It’s not supported by some older browsers (IE, use Babel to polyfill it), but works in modern ones. It looks like this:

```
    let options = {
    title: "Menu",
    height: 200,
    width: 100
};

// title = property named title
// rest = object with the rest of properties
let {title, ...rest} = options;

// now title="Menu", rest={height: 200, width: 100}
alert(rest.height); // 200
alert(rest.width);  // 100
```

Gotcha if there's no let

In the examples above variables were declared right in the assignment: `let {...} = {...}`. Of course, we could use existing variables too, without `let`. But there's a catch. This won't work:

```
    let title, width, height;

// error in this line
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

The problem is that JavaScript treats `{...}` in the main code flow (not inside another expression) as a code block. Such code blocks can be used to group statements, like this:

```
{  
  // a code block  
  let message = "Hello";  
  // ...  
  alert( message );  
}
```

So here JavaScript assumes that we have a code block, that's why there's an error. We want destructuring instead. To show JavaScript that it's not a code block, we can wrap the expression in parentheses (...):

```
let title, width, height;  
  
// okay now  
({title, width, height} = {title: "Menu", width: 200, height: 100});  
  
alert( title ); // Menu
```

Nested destructuringIf an object or an array contain other nested objects and arrays, we can use more complex left-side patterns to extract deeper portions. In the code below options has another object in the property size and an array in the property items. The pattern on the left side of the assignment has the same structure to extract values from them:

```
    let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// destructuring assignment split in multiple lines for clarity
let {
  size: { // put size here
    width,
    height
  },
  items: [item1, item2], // assign items here
  title = "Menu" // not present in the object (default value is used)
} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
alert(item1); // Cake
alert(item2); // Donut
```

All properties of options object except extra that is absent in the left part, are

assigned to corresponding variables:

Finally, we have width, height, item1, item2 and title from the default value. Note that there are no variables for size and items, as we take their content instead.

Smart function parameters

There are times when a function has many parameters, most of which are optional. That's especially true for user interfaces. Imagine a function that creates a menu. It may have a width, a height, a title, items list and so on. Here's a bad way to write such function:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {  
  // ...  
}
```

In real-life, the problem is how to remember the order of arguments. Usually IDEs try to help us, especially if the code is well-documented, but still... Another problem is how to call a function when most parameters are ok by default. Like this?

```
// undefined where default values are fine  
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

That's ugly. And becomes unreadable when we deal with more parameters.

Destructuring comes to the rescue!

We can pass parameters as an object, and the function immediately deconstructurizes them into variables:

```

        // we pass object to function
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// ...and it immediately expands it to variables
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
  // title, items – taken from options,
  // width, height – defaults used
  alert( `${title} ${width} ${height}` ); // My Menu 200 100
  alert( items ); // Item1, Item2
}

showMenu(options);

```

We can also use more complex destructuring with nested objects and colon mappings:

```

let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100, // width goes to w
  height: h = 200, // height goes to h

```



```

    items: [item1, item2] // items first element goes to item1, second to item2
  }) {
    alert( `${title} ${w} ${h}` ); // My Menu 100 200
    alert( item1 ); // Item1
    alert( item2 ); // Item2
  }

showMenu(options);

```

The full syntax is the same as for a destructuring assignment:

```

    function({
incomingProperty: varName = defaultValue
    ...
})

```

Then, for an object of parameters, there will be a variable `varName` for property `incomingProperty`, with `defaultValue` by default. Please note that such destructuring assumes that `showMenu()` does have an argument. If we want all values by default, then we should specify an empty object:

```

    showMenu({}); // ok, all values are default

showMenu(); // this would give an error

```

We can fix this by making `{}` the default value for the whole object of parameters:

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {  
  alert( `${title} ${width} ${height}` );  
}  
  
showMenu(); // Menu 100 200
```

In the code above, the whole arguments object is {} by default, so there's always something to destructure.

Summary

Destructuring assignment allows for instantly mapping an object or array onto many variables.

The full object syntax:

```
let {prop : varName = default, ...rest} = object
```

This means that property prop should go into the variable varName and, if no such property exists, then the default value should be used.
Object properties that have no mapping are copied to the rest object.

The full array syntax:

```
let [item1 = default, item2, ...rest] = array
```

The first item goes to item1; the second goes into item2, all the rest makes the array rest.

It's possible to extract data from nested arrays/objects, for that the left side must have the same structure as the right one.

TasksDestructuring assignmentimportance: 5We have an object:

```
let user = {  
  name: "John",  
  years: 30  
};
```

Write the destructuring assignment that reads:

name property into the variable name.

years property into the variable age.

isAdmin property into the variable isAdmin (false, if no such property)

Here's an example of the values after your assignment:

```
let user = { name: "John", years: 30 };  
  
// your code to the left side:  
// ... = user  
  
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

solution

```
    let user = {  
      name: "John",  
      years: 30  
    };
```

```
let {name, years: age, isAdmin = false} = user;
```

```
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

The maximal salaryimportance: 5There is a salaries object:

```
    let salaries = {  
      "John": 100,  
      "Pete": 300,  
      "Mary": 250  
    };
```

Create the function `topSalary(salaries)` that returns the name of the top-paid person.

If `salaries` is empty, it should return `null`.

If there are multiple top-paid persons, return any of them.

P.S. Use `Object.entries` and destructuring to iterate over key/value pairs.

Open a sandbox with tests.solution

```
function topSalary(salaries) {  
  
  let maxSalary = 0;  
  let maxName = null;
```

```

for(const [name, salary] of Object.entries(salaries)) {
  if (maxSalary < salary) {
    maxSalary = salary;
    maxName = name;
  }
}

return maxName;
}

```

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/destructuring-assignment","identifier":"\destructuring-assignment"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageData types{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Data types","item":"https://javascript.info/data-types"]}]October 14, 2022Date and timeLet's meet a new built-in object: Date. It stores the date, time and provides methods for date/time management. For instance, we can use it to store creation/modification times, to measure time, or just to print out the current date.

CreationTo create a new Date object call new Date() with one of the following arguments:

new Date()

Without arguments – create a Date object for the current date and time:

```
let now = new Date();  
alert( now ); // shows current date/time
```

```
new Date(milliseconds)
```

Create a Date object with the time equal to number of milliseconds (1/1000 of a second) passed after the Jan 1st of 1970 UTC+0.

```
// 0 means 01.01.1970 UTC+0  
let Jan01_1970 = new Date(0);  
alert( Jan01_1970 );  
  
// now add 24 hours, get 02.01.1970 UTC+0  
let Jan02_1970 = new Date(24 * 3600 * 1000);  
alert( Jan02_1970 );
```

An integer number representing the number of milliseconds that has passed since the beginning of 1970 is called a timestamp.

It's a lightweight numeric representation of a date. We can always create a date from a timestamp using `new Date(timestamp)` and convert the existing Date object to a timestamp using the `date.getTime()` method (see below).

Dates before 01.01.1970 have negative timestamps, e.g.:

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

```
new Date(datestring)
```

If there is a single argument, and it's a string, then it is parsed automatically. The algorithm is the same as `Date.parse` uses, we'll cover it later.

```
let date = new Date("2017-01-26");
alert(date);
// The time is not set, so it's assumed to be midnight GMT and
// is adjusted according to the timezone the code is run in
// So the result could be
// Thu Jan 26 2017 11:00:00 GMT+1100 (Australian Eastern Daylight Time)
// or
// Wed Jan 25 2017 16:00:00 GMT-0800 (Pacific Standard Time)
```

```
new Date(year, month, date, hours, minutes, seconds, ms)
```

Create the date with the given components in the local time zone. Only the first two arguments are obligatory.

The year should have 4 digits. For compatibility, 2 digits are also accepted and considered 19xx, e.g. 98 is the same as 1998 here, but always using 4 digits is strongly encouraged.

The month count starts with 0 (Jan), up to 11 (Dec).

The date parameter is actually the day of month, if absent then 1 is assumed.

If hours/minutes/seconds/ms is absent, they are assumed to be equal 0.

For instance:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // the same, hours etc are 0 by default
```

The maximal precision is 1 ms (1/1000 sec):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Access date components There are methods to access the year, month and so on from the Date object:

`getFullYear()`

Get the year (4 digits)

`getMonth()`

Get the month, from 0 to 11.

`getDate()`

Get the day of month, from 1 to 31, the name of the method does look a little bit strange.

`getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()`
Get the corresponding time components.

Not `getYear()`, but `getFullYear()`

Many JavaScript engines implement a non-standard method `getYear()`. This method is deprecated. It returns 2-digit year sometimes. Please never use it. There is `getFullYear()` for the year.

Additionally, we can get a day of week:

`getDay()`

Get the day of week, from 0 (Sunday) to 6 (Saturday). The first day is always Sunday, in some countries that's not so, but can't be changed.

All the methods above return the components relative to the local time zone.

There are also their UTC-counterparts, that return day, month, year and so on for the time zone UTC+0: `getUTCFullYear()`, `getUTCMonth()`, `getUTCDay()`. Just insert the "UTC" right after "get".

If your local time zone is shifted relative to UTC, then the code below shows different hours:

```
// current date
let date = new Date();

// the hour in your current time zone
alert( date.getHours() );

// the hour in UTC+0 time zone (London time without daylight savings)
alert( date.getUTCHours() );
```

Besides the given methods, there are two special ones that do not have a UTC-variant:

`getTime()`

Returns the timestamp for the date – a number of milliseconds passed from the January 1st of 1970 UTC+0.

`getTimezoneOffset()`

Returns the difference between UTC and the local time zone, in minutes:

```
// if you are in timezone UTC-1, outputs 60
// if you are in timezone UTC+3, outputs -180
alert( new Date().getTimezoneOffset() );
```

Setting date componentsThe following methods allow to set date/time components:

```
setFullYear(year, [month], [date])
setMonth(month, [date])
setDate(date)
setHours(hour, [min], [sec], [ms])
setMinutes(min, [sec], [ms])
setSeconds(sec, [ms])
setMilliseconds(ms)
setTime(milliseconds) (sets the whole date by milliseconds since 01.01.1970 UTC)
```

Every one of them except `setTime()` has a UTC-variant, for instance: `setUTCHours()`. As we can see, some methods can set multiple components at once, for example `setHours`. The components that are not mentioned are not modified. For instance:

```
let today = new Date();

today.setHours(0);
alert(today); // still today, but the hour is changed to 0

today.setHours(0, 0, 0, 0);
alert(today); // still today, now 00:00:00 sharp.
```

AutocorrectionThe autocorrection is a very handy feature of Date objects. We can set out-of-range values, and it will auto-adjust itself. For instance:

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 !?!
alert(date); // ...is 1st Feb 2013!
```

Out-of-range date components are distributed automatically. Let's say we need to increase the date "28 Feb 2016" by 2 days. It may be "2 Mar" or "1 Mar" in case of a leap-year. We don't need to think about it. Just add 2 days. The Date object will do the rest:

```
let date = new Date(2016, 1, 28);  
date.setDate(date.getDate() + 2);  
  
alert( date ); // 1 Mar 2016
```

That feature is often used to get the date after the given period of time. For instance, let's get the date for "70 seconds after now":

```
let date = new Date();  
date.setSeconds(date.getSeconds() + 70);  
  
alert( date ); // shows the correct date
```

We can also set zero or even negative values. For example:

```
let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // set day 1 of month
alert( date );

date.setDate(0); // min day is 1, so the last day of the previous month is assumed
alert( date ); // 31 Dec 2015
```

Date to number, date diffWhen a Date object is converted to number, it becomes the timestamp same as `date.getTime()`:

```
let date = new Date();
alert(+date); // the number of milliseconds, same as date.getTime()
```

The important side effect: dates can be subtracted, the result is their difference in ms.
That can be used for time measurements:

```
    let start = new Date(); // start measuring time

// do the job
for (let i = 0; i < 100000; i++) {
    let doSomething = i * i * i;
}

let end = new Date(); // end measuring time

alert( `The loop took ${end - start} ms` );
```

`Date.now()` If we only want to measure time, we don't need the `Date` object. There's a special method `Date.now()` that returns the current timestamp. It is semantically equivalent to `new Date().getTime()`, but it doesn't create an intermediate `Date` object. So it's faster and doesn't put pressure on garbage collection. It is used mostly for convenience or when performance matters, like in games in JavaScript or other specialized applications. So this is probably better:

```
    let start = Date.now(); // milliseconds count from 1 Jan 1970

// do the job
for (let i = 0; i < 100000; i++) {
    let doSomething = i * i * i;
}

let end = Date.now(); // done

alert( `The loop took ${end - start} ms` ); // subtract numbers, not dates
```

BenchmarkingIf we want a reliable benchmark of CPU-hungry function, we should be careful.

For instance, let's measure two functions that calculate the difference between two dates: which one is faster?

Such performance measurements are often called "benchmarks".

```
// we have date1 and date2, which function faster returns their difference in ms?
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// or
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

These two do exactly the same thing, but one of them uses an explicit `date.getTime()` to get the date in ms, and the other one relies on a date-to-number transform. Their result is always the same.

So, which one is faster?

The first idea may be to run them many times in a row and measure the time difference.

For our case, functions are very simple, so we have to do it at least 100000 times.

Let's measure:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}
```

```

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert( 'Time of diffSubtract: ' + bench(diffSubtract) + 'ms' );
alert( 'Time of diffGetTime: ' + bench(diffGetTime) + 'ms' );

```

Wow! Using `getTime()` is so much faster! That's because there's no type conversion, it is much easier for engines to optimize.

Okay, we have something. But that's not a good benchmark yet. Imagine that at the time of running `bench(diffSubtract)` CPU was doing something in parallel, and it was taking resources. And by the time of running `bench(diffGetTime)` that work has finished.

A pretty real scenario for a modern multi-process OS.

As a result, the first benchmark will have less CPU resources than the second. That may lead to wrong results.

For more reliable benchmarking, the whole pack of benchmarks should be rerun multiple times.

For example, like this:

```

function diffSubtract(date1, date2) {
  return date2 - date1;
}

```



```

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// run bench(diffSubtract) and bench(diffGetTime) each 10 times alternating
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

alert( 'Total time for diffSubtract: ' + time1 );
alert( 'Total time for diffGetTime: ' + time2 );

```

Modern JavaScript engines start applying advanced optimizations only to “hot code” that executes many times (no need to optimize rarely executed things). So, in the example above, first executions are not well-optimized. We may want to add a heat-up run:

```

    // added for "heating up" prior to the main loop
    bench(diffSubtract);
    bench(diffGetTime);

    // now benchmark
    for (let i = 0; i < 10; i++) {
      time1 += bench(diffSubtract);
      time2 += bench(diffGetTime);
    }

```

Be careful doing microbenchmarking

Modern JavaScript engines perform many optimizations. They may tweak results of “artificial tests” compared to “normal usage”, especially when we benchmark something very small, such as how an operator works, or a built-in function. So if you seriously want to understand performance, then please study how the JavaScript engine works. And then you probably won't need microbenchmarks at all. The great pack of articles about V8 can be found at <https://mrale.ph>.

Date.parse from a string The method `Date.parse(str)` can read a date from a string. The string format should be: `YYYY-MM-DDTHH:mm:ss.sssZ`, where:

`YYYY-MM-DD` – is the date: year-month-day.

The character "T" is used as the delimiter.

`HH:mm:ss.sss` – is the time: hours, minutes, seconds and milliseconds.

The optional 'Z' part denotes the time zone in the format `+-hh:mm`. A single letter Z would mean UTC+0.

Shorter variants are also possible, like `YYYY-MM-DD` or `YYYY-MM` or even `YYYY`.

The call to `Date.parse(str)` parses the string in the given format and returns the timestamp (number of milliseconds from 1 Jan 1970 UTC+0). If the format is invalid, returns NaN.

For instance:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');  
  
alert(ms); // 1327611110417 (timestamp)
```

We can instantly create a new Date object from the timestamp:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );  
  
alert(date);
```

Summary

Date and time in JavaScript are represented with the Date object. We can't create "only date" or "only time": Date objects always carry both.

Months are counted from zero (yes, January is a zero month).

Days of week in `getDay()` are also counted from zero (that's Sunday).

Date auto-corrects itself when out-of-range components are set. Good for adding/subtracting days/months/hours.

Dates can be subtracted, giving their difference in milliseconds. That's because a Date becomes the timestamp when converted to a number.

Use `Date.now()` to get the current timestamp fast.

Note that unlike many other systems, timestamps in JavaScript are in milliseconds, not in seconds.

Sometimes we need more precise time measurements. JavaScript itself does not have a way to measure time in microseconds (1 millionth of a second), but most environments provide it. For instance, browser has `performance.now()` that gives the number of milliseconds from the start of page loading with microsecond precision (3 digits after the point):

```
alert(`Loading started ${performance.now()}ms ago`);
```

```
// Something like: "Loading started 34731.26000000001ms ago"
// .26 is microseconds (260 microseconds)
// more than 3 digits after the decimal point are precision errors, only the first 3 are correct
```

Node.js has microtime module and other ways. Technically, almost any device and environment allows to get more precision, it's just not in Date.

TasksCreate a dateimportance: 5Create a Date object for the date: Feb 20, 2012, 3:12am. The time zone is local.

Show it using alert.

solutionThe new Date constructor uses the local time zone. So the only important thing to remember is that months start from zero.

So February has number 1.

Here's an example with numbers as date components:

```
//new Date(year, month, date, hour, minute, second, millisecond)
let d1 = new Date(2012, 1, 20, 3, 12);
alert( d1 );
```

We could also create a date from a string, like this:

```
//new Date(datastring)
let d2 = new Date("2012-02-20T03:12");
alert( d2 );
```

Show a weekdayimportance: 5Write a function `getWeekDay(date)` to show the weekday in short format: 'MO', 'TU', 'WE', 'TH', 'FR', 'SA', 'SU'.
For instance:

```
let date = new Date(2012, 0, 3); // 3 Jan 2012
alert( getWeekDay(date) );      // should output "TU"
```

Open a sandbox with tests.solutionThe method `date.getDay()` returns the number of the weekday, starting from sunday.
Let's make an array of weekdays, so that we can get the proper day name by its number:

```
function getWeekDay(date) {
let days = ['SU', 'MO', 'TU', 'WE', 'TH', 'FR', 'SA'];

return days[date.getDay()];
}

let date = new Date(2014, 0, 3); // 3 Jan 2014
alert( getWeekDay(date) ); // FR
```

Open the solution with tests in a sandbox. European weekday importance: 5 European countries have days of week starting with Monday (number 1), then Tuesday (number 2) and till Sunday (number 7). Write a function `getLocalDay(date)` that returns the “European” day of week for date.

```
let date = new Date(2012, 0, 3); // 3 Jan 2012
alert( getLocalDay(date) );      // tuesday, should show 2
```

Open a sandbox with tests.

```
function getLocalDay(date) {

let day = date.getDay();

if (day == 0) { // weekday 0 (sunday) is 7 in european
    day = 7;
}

return day;
}
```

Open the solution with tests in a sandbox. Which day of month was many days ago? importance: 4 Create a function `getDateAgo(date, days)` to return the day of month days ago from the date.

For instance, if today is 20th, then `getDateAgo(new Date(), 1)` should be 19th and `getDateAgo(new Date(), 2)` should be 18th. Should work reliably for days=365 or more:

```
let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

P.S. The function should not modify the given date.
Open a sandbox with tests.solutionThe idea is simple: to subtract given number of days from date:

```
function getDateAgo(date, days) {  
  date.setDate(date.getDate() - days);  
  return date.getDate();  
}
```

...But the function should not change date. That's an important thing, because the outer code which gives us the date does not expect it to change.
To implement it let's clone the date, like this:

```
function getDateAgo(date, days) {  
  let dateCopy = new Date(date);  
  
  dateCopy.setDate(date.getDate() - days);  
  return dateCopy.getDate();  
}  
  
let date = new Date(2015, 0, 2);  
  
alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)  
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)  
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

Open the solution with tests in a sandbox. Last day of month? importance: 5 Write a function `getLastDayOfMonth(year, month)` that returns the last day of month. Sometimes it is 30th, 31st or even 28/29th for Feb. Parameters:

year – four-digits year, for instance 2012.
month – month, from 0 to 11.

For instance, `getLastDayOfMonth(2012, 1) = 29` (leap year, Feb).

Open a sandbox with tests. solution Let's create a date using the next month, but pass zero as the day:

```
function getLastDayOfMonth(year, month) {  
  let date = new Date(year, month + 1, 0);  
  return date.getDate();  
}
```

```
alert( getLastDayOfMonth(2012, 0) ); // 31  
alert( getLastDayOfMonth(2012, 1) ); // 29  
alert( getLastDayOfMonth(2013, 1) ); // 28
```

Normally, dates start from 1, but technically we can pass any number, the date will autoadjust itself. So when we pass 0, then it means “one day before 1st day of the month”, in other words: “the last day of the previous month”.

Open the solution with tests in a sandbox. How many seconds have passed today? importance: 5 Write a function `getSecondsToday()` that returns the number of seconds from the beginning of today.

For instance, if now were 10:00 am, and there was no daylight savings shift, then:

```
getSecondsToday() == 36000 // (3600 * 10)
```


The function should work in any day. That is, it should not have a hard-coded value of “today”.

solutionTo get the number of seconds, we can generate a date using the current day and time 00:00:00, then subtract it from “now”.

The difference is the number of milliseconds from the beginning of the day, that we should divide by 1000 to get seconds:

```
function getSecondsToday() {  
  let now = new Date();  
  
  // create an object using the current day/month/year  
  let today = new Date(now.getFullYear(), now.getMonth(), now.getDate());  
  
  let diff = now - today; // ms difference  
  return Math.round(diff / 1000); // make seconds  
}  
  
alert( getSecondsToday() );
```

An alternative solution would be to get hours/minutes/seconds and convert them to seconds:

```
function getSecondsToday() {  
  let d = new Date();  
  return d.getHours() * 3600 + d.getMinutes() * 60 + d.getSeconds();  
}  
  
alert( getSecondsToday() );
```

How many seconds till tomorrow?importance: 5Create a function `getSecondsToTomorrow()` that returns the number of seconds till tomorrow. For instance, if now is 23:00, then:

```
getSecondsToTomorrow() == 3600
```

P.S. The function should work at any day, the “today” is not hardcoded.
solutionTo get the number of milliseconds till tomorrow, we can from “tomorrow 00:00:00” subtract the current date.
First, we generate that “tomorrow”, and then do it:

```
function getSecondsToTomorrow() {  
  let now = new Date();  
  
  // tomorrow date  
  let tomorrow = new Date(now.getFullYear(), now.getMonth(), now.getDate()+1);  
  
  let diff = tomorrow - now; // difference in ms
```

```
return Math.round(diff / 1000); // convert to seconds
}
```

Alternative solution:

```
function getSecondsToTomorrow() {
let now = new Date();
let hour = now.getHours();
let minutes = now.getMinutes();
let seconds = now.getSeconds();
let totalSecondsToday = (hour * 60 + minutes) * 60 + seconds;
let totalSecondsInADay = 86400;

return totalSecondsInADay - totalSecondsToday;
}
```

Please note that many countries have Daylight Savings Time (DST), so there may be days with 23 or 25 hours. We may want to treat such days separately.

Format the relative dateimportance: 4Write a function `formatDate(date)` that should format date as follows:

If since date passed less than 1 second, then "right now".

Otherwise, if since date passed less than 1 minute, then "n sec. ago".

Otherwise, if less than an hour, then "m min. ago".

Otherwise, the full date in the format "DD.MM.YY HH:mm". That is: "day.month.year hours:minutes", all in 2-digit format, e.g. 31.12.16 10:00.

For instance:

```

    alert( formatDate(new Date(new Date - 1)) ); // "right now"

alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 sec. ago"

alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 min. ago"

// yesterday's date like 31.12.16 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );

```

Open a sandbox with tests.solutionTo get the time from date till now – let's subtract the dates.

```

function formatDate(date) {
let diff = new Date() - date; // the difference in milliseconds

if (diff < 1000) { // less than 1 second
    return 'right now';
}

let sec = Math.floor(diff / 1000); // convert diff to seconds

if (sec < 60) {
    return sec + ' sec. ago';
}

let min = Math.floor(diff / 60000); // convert diff to minutes
if (min < 60) {
    return min + ' min. ago';
}

// format the date
// add leading zeroes to single-digit day/month/hours/minutes

```

```

let d = date;
d = [
  '0' + d.getDate(),
  '0' + (d.getMonth() + 1),
  '' + d.getFullYear(),
  '0' + d.getHours(),
  '0' + d.getMinutes()
].map(component => component.slice(-2)); // take last 2 digits of every component

// join the components into date
return d.slice(0, 3).join('.') + ' ' + d.slice(3).join(':');
}

alert( formatDate(new Date(new Date - 1)) ); // "right now"

alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 sec. ago"

alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 min. ago"

// yesterday's date like 31.12.2016 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );

```

Alternative solution:

```

function formatDate(date) {
  let dayOfMonth = date.getDate();
  let month = date.getMonth() + 1;
  let year = date.getFullYear();
  let hour = date.getHours();
  let minutes = date.getMinutes();
  let diffMs = new Date() - date;
  let diffSec = Math.round(diffMs / 1000);
  let diffMin = diffSec / 60;

```

```

let diffHour = diffMin / 60;

// formatting
year = year.toString().slice(-2);
month = month < 10 ? '0' + month : month;
dayOfMonth = dayOfMonth < 10 ? '0' + dayOfMonth : dayOfMonth;
hour = hour < 10 ? '0' + hour : hour;
minutes = minutes < 10 ? '0' + minutes : minutes;

if (diffSec < 1) {
  return 'right now';
} else if (diffMin < 1) {
  return `${diffSec} sec. ago`;
} else if (diffHour < 1) {
  return `${diffMin} min. ago`;
} else {
  return `${dayOfMonth}.${month}.${year} ${hour}:${minutes}`;
}
}

```

Open the solution with tests in a sandbox. Ctrl + ! • Previous lesson Ctrl + ! • Next lesson

Share Tutorial map Comments read this before commenting... If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/date", "identifier": "Vdate" }); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true;

Tutorial The JavaScript language Data types { "@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{ "@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/" }, { "@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js" }, { "@type": "ListItem", "position": 3, "name": "Data types", "item": "https://javascript.info/data-types" }] }

October 14, 2022 JSON methods, toJSON Let's say we have a complex object, and we'd like to convert it into a string, to send it over a network, or just to output it for logging purposes. Naturally, such a string should include all important properties. We could implement the conversion like this:

```
    let user = {
name: "John",
age: 30,

toString() {
    return `${name: "${this.name}", age: ${this.age}}`;
}
};

alert(user); // {name: "John", age: 30}
```

...But in the process of development, new properties are added, old properties are renamed and removed. Updating such toString every time can become a pain. We could try to loop over properties in it, but what if the object is complex and has nested objects in properties? We'd need to implement their conversion as well. Luckily, there's no need to write the code to handle all this. The task has been solved already.

JSON.stringifyThe JSON (JavaScript Object Notation) is a general format to represent values and objects. It is described as in RFC 4627 standard. Initially it was made for JavaScript, but many other languages have libraries to handle it as well. So it's easy to use JSON for data exchange when the client uses JavaScript and the server is written on Ruby/PHP/Java/Whatever.

JavaScript provides methods:

JSON.stringify to convert objects into JSON.

JSON.parse to convert JSON back into an object.

For instance, here we JSON.stringify a student:

```

    let student = {
    name: 'John',
    age: 30,
    isAdmin: false,
    courses: ['html', 'css', 'js'],
    spouse: null
    };

let json = JSON.stringify(student);

alert(typeof json); // we've got a string!

alert(json);
/* JSON-encoded object:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "spouse": null
}
*/

```

The method `JSON.stringify(student)` takes the object and converts it into a string. The resulting json string is called a JSON-encoded or serialized or stringified or marshalled object. We are ready to send it over the wire or put into a plain data store. Please note that a JSON-encoded object has several important differences from the object literal:

Strings use double quotes. No single quotes or backticks in JSON. So `'John'` becomes `"John"`.

Object property names are double-quoted also. That's obligatory. So `age:30` becomes `"age":30`.

`JSON.stringify` can be applied to primitives as well.
JSON supports following data types:

Objects { ... }
Arrays [...]
Primitives:

strings,
numbers,
boolean values true/false,
null.

For instance:

```
// a number in JSON is just a number  
alert( JSON.stringify(1) ) // 1
```

```
// a string in JSON is still a string, but double-quoted  
alert( JSON.stringify('test') ) // "test"
```

```
alert( JSON.stringify(true) ); // true
```

```
alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON is data-only language-independent specification, so some JavaScript-specific object properties are skipped by `JSON.stringify`.

Namely:

Function properties (methods).

Symbolic keys and values.

Properties that store undefined.

```
    let user = {
    sayHi() { // ignored
      alert("Hello");
    },
    [Symbol("id")]: 123, // ignored
    something: undefined // ignored
  };

  alert( JSON.stringify(user) ); // {} (empty object)
```

Usually that's fine. If that's not what we want, then soon we'll see how to customize the process.

The great thing is that nested objects are supported and converted automatically. For instance:

```
    let meetup = {
    title: "Conference",
    room: {
      number: 23,
      participants: ["john", "ann"]
    }
  };

  alert( JSON.stringify(meetup) );
  /* The whole structure is stringified:
  {
    "title":"Conference",
```

```
"room":{"number":23,"participants":["john","ann"]},
}
*/
```

The important limitation: there must be no circular references.
For instance:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};

meetup.place = room;    // meetup references room
room.occupiedBy = meetup; // room references meetup

JSON.stringify(meetup); // Error: Converting circular structure to JSON
```

Here, the conversion fails, because of circular reference: room.occupiedBy references meetup, and meetup.place references room:

Excluding and transforming: replacerThe full syntax of JSON.stringify is:

```
let json = JSON.stringify(value[, replacer, space])
```

value

A value to encode.

replacer

Array of properties to encode or a mapping function function(key, value).

space

Amount of space to use for formatting

Most of the time, JSON.stringify is used with the first argument only. But if we need to fine-tune the replacement process, like to filter out circular references, we can use the second argument of JSON.stringify.

If we pass an array of properties to it, only these properties will be encoded.

For instance:

```
let room = {  
  number: 23  
};
```

```
let meetup = {  
  title: "Conference",  
  participants: [{name: "John"}, {name: "Alice"}],  
  place: room // meetup references room  
};
```

```
room.occupiedBy = meetup; // room references meetup
```

```
alert( JSON.stringify(meetup, ['title', 'participants']) );  
// {"title":"Conference","participants":[{"name":"John"}, {"name":"Alice"}]}
```

Here we are probably too strict. The property list is applied to the whole object structure. So the objects in participants are empty, because name is not in the list. Let's include in the list every property except room.occupiedBy that would cause the circular reference:

```
    let room = {
    number: 23
};

let meetup = {
    title: "Conference",
    participants: [{name: "John"}, {name: "Alice"}],
    place: room // meetup references room
};

room.occupiedBy = meetup; // room references meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );
/*
{
  "title":"Conference",
  "participants":[{"name":"John"}, {"name":"Alice"}],
  "place":{"number":23}
}
*/
```

Now everything except occupiedBy is serialized. But the list of properties is quite long. Fortunately, we can use a function instead of an array as the replacer. The function will be called for every (key, value) pair and should return the “replaced” value, which will be used instead of the original one. Or undefined if the value is to be skipped.

In our case, we can return value “as is” for everything except occupiedBy. To ignore occupiedBy, the code below returns undefined:

```
    let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup references room
};

room.occupiedBy = meetup; // room references meetup

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert( `${key}: ${value}` );
  return (key == 'occupiedBy') ? undefined : value;
})));

/* key:value pairs that come to replacer:
:      [object Object]
title:  Conference
participants: [object Object],[object Object]
0:      [object Object]
name:    John
1:      [object Object]
name:    Alice
place:   [object Object]
number:  23
occupiedBy: [object Object]
*/
```

Please note that replacer function gets every key/value pair including nested objects and array items. It is applied recursively. The value of this inside replacer is the object that contains the current property.

The first call is special. It is made using a special “wrapper object”: {“”: meetup}. In other words, the first (key, value) pair has an empty key, and the value is the target object as a whole. That’s why the first line is “[object Object]” in the example above.

The idea is to provide as much power for replacer as possible: it has a chance to analyze and replace/skip even the whole object if necessary.

Formatting: spaceThe third argument of JSON.stringify(value, replacer, space) is the number of spaces to use for pretty formatting.

Previously, all stringified objects had no indents and extra spaces. That’s fine if we want to send an object over a network. The space argument is used exclusively for a nice output.

Here space = 2 tells JavaScript to show nested objects on multiple lines, with indentation of 2 spaces inside an object:

```
    let user = {
name: "John",
age: 25,
roles: {
  isAdmin: false,
  isEditor: true
}
};

alert(JSON.stringify(user, null, 2));
/* two-space indents:
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
```

```
*/
```

/* for JSON.stringify(user, null, 4) the result would be more indented:

```
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/
```

The third argument can also be a string. In this case, the string is used for indentation instead of a number of spaces.

The space parameter is used solely for logging and nice-output purposes.

Custom “toJSON” Like toString for string conversion, an object may provide method toJSON for to-JSON conversion. JSON.stringify automatically calls it if available.

For instance:

```
    let room = {
      number: 23
    };

    let meetup = {
      title: "Conference",
      date: new Date(Date.UTC(2017, 0, 1)),
      room
    };

    alert( JSON.stringify(meetup) );
    /*
    {
```



```

    "title": "Conference",
    "date": "2017-01-01T00:00:00.000Z", // (1)
    "room": {"number": 23}           // (2)
  }
*/

```

Here we can see that date (1) became a string. That's because all dates have a built-in toJSON method which returns such kind of string. Now let's add a custom toJSON for our object room (2):

```

    let room = {
      number: 23,
      toJSON() {
        return this.number;
      }
    };

    let meetup = {
      title: "Conference",
      room
    };

    alert( JSON.stringify(room) ); // 23

    alert( JSON.stringify(meetup) );
    /*
    {
      "title": "Conference",
      "room": 23
    }
    */

```

As we can see, `toJSON` is used both for the direct call `JSON.stringify(room)` and when `room` is nested in another encoded object. `JSON.parse` To decode a JSON-string, we need another method named `JSON.parse`. The syntax:

```
let value = JSON.parse(str, [reviver]);
```

`str`
JSON-string to parse.
`reviver`
Optional function(key,value) that will be called for each (key, value) pair and can transform the value.

For instance:

```
// stringified array
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

Or for nested objects:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';  
  
let user = JSON.parse(userData);  
  
alert( user.friends[1] ); // 1
```

The JSON may be as complex as necessary, objects and arrays can include other objects and arrays. But they must obey the same JSON format. Here are typical mistakes in hand-written JSON (sometimes we have to write it for debugging purposes):

```
let json = `{  
  name: "John",           // mistake: property name without quotes  
  "surname": 'Smith',     // mistake: single quotes in value (must be double)  
  'isAdmin': false       // mistake: single quotes in key (must be double)  
  "birthday": new Date(2000, 2, 3), // mistake: no "new" is allowed, only bare values  
  "friends": [0,1,2,3]    // here all fine  
}`;
```

Besides, JSON does not support comments. Adding a comment to JSON makes it invalid.

There's another format named JSON5, which allows unquoted keys, comments etc. But this is a standalone library, not in the specification of the language.

The regular JSON is that strict not because its developers are lazy, but to allow easy, reliable and very fast implementations of the parsing algorithm.

Using `reviver` Imagine, we got a stringified meetup object from the server.

It looks like this:

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

...And now we need to deserialize it, to turn back into JavaScript object.
Let's do it by calling JSON.parse:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

Whoops! An error!
The value of meetup.date is a string, not a Date object. How could JSON.parse know that it should transform that string into a Date?
Let's pass to JSON.parse the reviving function as the second argument, that returns all values "as is", but date will become a Date:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

```
let meetup = JSON.parse(str, function(key, value) {  
  if (key == 'date') return new Date(value);  
  return value;  
});
```

```
alert( meetup.date.getDate() ); // now works!
```

By the way, that works for nested objects as well:

```
let schedule = `{  
  "meetups": [  
    {"title": "Conference", "date": "2017-11-30T12:00:00.000Z"},  
    {"title": "Birthday", "date": "2017-04-18T12:00:00.000Z"}  
  ]  
}`;
```

```
schedule = JSON.parse(schedule, function(key, value) {  
  if (key == 'date') return new Date(value);  
  return value;  
});
```

```
alert( schedule.meetups[1].date.getDate() ); // works!
```

Summary

JSON is a data format that has its own independent standard and libraries for most programming languages.

JSON supports plain objects, arrays, strings, numbers, booleans, and null.

JavaScript provides methods `JSON.stringify` to serialize into JSON and `JSON.parse` to read from JSON.

Both methods support transformer functions for smart reading/writing.

If an object has `toJSON`, then it is called by `JSON.stringify`.

Tasks Turn the object into JSON and back importance: 5 Turn the user into JSON and then read it back into another variable.

```
let user = {  
  name: "John Smith",  
  age: 35  
};
```

solution

```
let user = {  
  name: "John Smith",  
  age: 35  
};
```

```
let user2 = JSON.parse(JSON.stringify(user));
```

Exclude backreferences importance: 5 In simple cases of circular references, we can exclude an offending property from serialization by its name.

But sometimes we can't just use the name, as it may be used both in circular references and normal properties. So we can check the property by its value.

Write replacer function to stringify everything, but remove properties that reference meetup:

```

    let room = {
    number: 23
};

let meetup = {
  title: "Conference",
  occupiedBy: [{name: "John"}, {name: "Alice"}],
  place: room
};

// circular references
room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  /* your code */
}));

/* result should be:
{
  "title":"Conference",
  "occupiedBy":[{"name":"John"}, {"name":"Alice"}],
  "place":{"number":23}
}
*/

```

solution

```

    let room = {
    number: 23
};

let meetup = {
  title: "Conference",

```

```

    occupiedBy: [{name: "John"}, {name: "Alice"}],
    place: room
  };

room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {
  return (key !== "" && value === meetup) ? undefined : value;
}));

/*
{
  "title": "Conference",
  "occupiedBy": [{"name": "John"}, {"name": "Alice"}],
  "place": {"number": 23}
}
*/

```

Here we also need to test `key===""` to exclude the first call where it is normal that value is `meetup`.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://v.javascript.info/vjson","identifier":"vjson"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript language{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}]}Advanced working with functionsRecursion and stackRest parameters and spread syntaxVariable scope, closureThe old "var"Global objectFunction object, NFEThe "new Function" syntaxScheduling: setTimeout and setIntervalDecorators and forwarding, call/applyFunction bindingArrow functions revisitedCtrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapTutorialThe JavaScript languageAdvanced working with functions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},{"@type":"ListItem","position":3,"name":"Advanced working with functions","item":"https://javascript.info/advanced-functions"}]}October 1,

2022Recursion and stackLet's return to functions and study them more in-depth.

Our first topic will be recursion.

If you are not new to programming, then it is probably familiar and you could skip this chapter.

Recursion is a programming pattern that is useful in situations when a task can be naturally split into several tasks of the same kind, but simpler. Or when a task can be simplified into an easy action plus a simpler variant of the same task. Or, as we'll see soon, to deal with certain data structures.

When a function solves a task, in the process it can call many other functions. A partial case of this is when a function calls itself. That's called recursion.

Two ways of thinkingFor something simple to start with – let's write a function `pow(x, n)` that raises `x` to a natural power of `n`. In other words, multiplies `x` by itself `n` times.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

There are two ways to implement it.

Iterative thinking: the for loop:

```
function pow(x, n) {
  let result = 1;

  // multiply result by x n times in the loop
  for (let i = 0; i < n; i++) {
    result *= x;
  }
}
```

```

    return result;
}

alert( pow(2, 3) ); // 8

```

Recursive thinking: simplify the task and call self:

```

    function pow(x, n) {
    if (n == 1) {
        return x;
    } else {
        return x * pow(x, n - 1);
    }
    }

alert( pow(2, 3) ); // 8

```

Please note how the recursive variant is fundamentally different. When `pow(x, n)` is called, the execution splits into two branches:

$$\text{pow}(x, n) = \begin{matrix} / \\ \text{if } n==1 \text{ } = x \\ \backslash \end{matrix}$$

```
else    = x * pow(x, n - 1)
```

If $n == 1$, then everything is trivial. It is called the base of recursion, because it immediately produces the obvious result: $\text{pow}(x, 1)$ equals x .

Otherwise, we can represent $\text{pow}(x, n)$ as $x * \text{pow}(x, n - 1)$. In maths, one would write $x^n = x * x^{n-1}$. This is called a recursive step: we transform the task into a simpler action (multiplication by x) and a simpler call of the same task (pow with lower n). Next steps simplify it further and further until n reaches 1.

We can also say that pow recursively calls itself till $n == 1$.

For example, to calculate $\text{pow}(2, 4)$ the recursive variant does these steps:

```
pow(2, 4) = 2 * pow(2, 3)
pow(2, 3) = 2 * pow(2, 2)
pow(2, 2) = 2 * pow(2, 1)
pow(2, 1) = 2
```

So, the recursion reduces a function call to a simpler one, and then – to even more simpler, and so on, until the result becomes obvious.

Recursion is usually shorter

A recursive solution is usually shorter than an iterative one.

Here we can rewrite the same using the conditional operator `?` instead of `if` to make $\text{pow}(x, n)$ more terse and still very readable:

```
function pow(x, n) {
return (n == 1) ? x : (x * pow(x, n - 1));
```

}

The maximal number of nested calls (including the first one) is called recursion depth. In our case, it will be exactly n .

The maximal recursion depth is limited by JavaScript engine. We can rely on it being 10000, some engines allow more, but 100000 is probably out of limit for the majority of them. There are automatic optimizations that help alleviate this ("tail calls optimizations"), but they are not yet supported everywhere and work only in simple cases.

That limits the application of recursion, but it still remains very wide. There are many tasks where recursive way of thinking gives simpler code, easier to maintain.

The execution context and stackNow let's examine how recursive calls work. For that we'll look under the hood of functions.

The information about the process of execution of a running function is stored in its execution context.

The execution context is an internal data structure that contains details about the execution of a function: where the control flow is now, the current variables, the value of this (we don't use it here) and few other internal details.

One function call has exactly one execution context associated with it.

When a function makes a nested call, the following happens:

The current function is paused.

The execution context associated with it is remembered in a special data structure called execution context stack.

The nested call executes.

After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.

Let's see what happens during the `pow(2, 3)` call.

`pow(2, 3)`In the beginning of the call `pow(2, 3)` the execution context will store variables: $x = 2$, $n = 3$, the execution flow is at line 1 of the function.

We can sketch it as:

```
Context: { x: 2, n: 3, at line 1 }  
pow(2, 3)
```

That's when the function starts to execute. The condition `n == 1` is falsy, so the flow continues into the second branch of if:

```

    function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

```

```

alert( pow(2, 3) );

```

The variables are same, but the line changes, so the context is now:

```

Context: { x: 2, n: 3, at line 5 }
pow(2, 3)

```

To calculate $x * \text{pow}(x, n - 1)$, we need to make a subcall of pow with new arguments pow(2, 2).

pow(2, 2) To do a nested call, JavaScript remembers the current execution context in the execution context stack.

Here we call the same function pow, but it absolutely doesn't matter. The process is the same for all functions:

The current context is “remembered” on top of the stack.

The new context is created for the subcall.

When the subcall is finished – the previous context is popped from the stack, and its execution continues.

Here's the context stack when we entered the subcall pow(2, 2):

```

Context: { x: 2, n: 2, at line 1 }
pow(2, 2)

```

Context: { x: 2, n: 3, at line 5 }
pow(2, 3)

The new current execution context is on top (and bold), and previous remembered contexts are below.

When we finish the subcall – it is easy to resume the previous context, because it keeps both variables and the exact place of the code where it stopped.

Please note:

Here in the picture we use the word “line”, as in our example there’s only one subcall in line, but generally a single line of code may contain multiple subcalls, like `pow(...) + pow(...) + somethingElse(...)`.

So it would be more precise to say that the execution resumes “immediately after the subcall”.

pow(2, 1)The process repeats: a new subcall is made at line 5, now with arguments `x=2, n=1`.

A new execution context is created, the previous one is pushed on top of the stack:

Context: { x: 2, n: 1, at line 1 }
pow(2, 1)

Context: { x: 2, n: 2, at line 5 }
pow(2, 2)

Context: { x: 2, n: 3, at line 5 }
pow(2, 3)

There are 2 old contexts now and 1 currently running for `pow(2, 1)`.

The exitDuring the execution of `pow(2, 1)`, unlike before, the condition `n == 1` is truthy, so the first branch of if works:

```
function pow(x, n) {  
  if (n == 1) {  
    return x;  
  } else {  
    return x * pow(x, n - 1);  
  }  
}
```

```
}  
}
```

There are no more nested calls, so the function finishes, returning 2.
As the function finishes, its execution context is not needed anymore, so it's removed from the memory. The previous one is restored off the top of the stack:

```
Context: { x: 2, n: 2, at line 5 }  
pow(2, 2)
```

```
Context: { x: 2, n: 3, at line 5 }  
pow(2, 3)
```

The execution of `pow(2, 2)` is resumed. It has the result of the subcall `pow(2, 1)`, so it also can finish the evaluation of `x * pow(x, n - 1)`, returning 4.
Then the previous context is restored:

```
Context: { x: 2, n: 3, at line 5 }  
pow(2, 3)
```

When it finishes, we have a result of `pow(2, 3) = 8`.

The recursion depth in this case was: 3.

As we can see from the illustrations above, recursion depth equals the maximal number of context in the stack.

Note the memory requirements. Contexts take memory. In our case, raising to the power of `n` actually requires the memory for `n` contexts, for all lower values of `n`.

A loop-based algorithm is more memory-saving:

```
function pow(x, n) {  
  let result = 1;  
  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  
  return result;  
}
```

```
}
```

The iterative pow uses a single context changing i and result in the process. Its memory requirements are small, fixed and do not depend on n. Any recursion can be rewritten as a loop. The loop variant usually can be made more effective.

...But sometimes the rewrite is non-trivial, especially when a function uses different recursive subcalls depending on conditions and merges their results or when the branching is more intricate. And the optimization may be unneeded and totally not worth the efforts.

Recursion can give a shorter code, easier to understand and support. Optimizations are not required in every place, mostly we need a good code, that's why it's used.

Recursive traversals Another great application of the recursion is a recursive traversal. Imagine, we have a company. The staff structure can be presented as an object:

```
let company = {
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 1600
  }],
  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],
    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```


In other words, a company has departments.

A department may have an array of staff. For instance, sales department has 2 employees: John and Alice.

Or a department may split into subdepartments, like development has two branches: sites and internals. Each of them has their own staff.

It is also possible that when a subdepartment grows, it divides into subsubdepartments (or teams).

For instance, the sites department in the future may be split into teams for siteA and siteB. And they, potentially, can split even more. That's not on the picture, just something to have in mind.

Now let's say we want a function to get the sum of all salaries. How can we do that? An iterative approach is not easy, because the structure is not simple. The first idea may be to make a for loop over company with nested subloop over 1st level departments. But then we need more nested subloops to iterate over the staff in 2nd level departments like sites... And then another subloop inside those for 3rd level departments that might appear in the future? If we put 3-4 nested subloops in the code to traverse a single object, it becomes rather ugly.

Let's try recursion.

As we can see, when our function gets a department to sum, there are two possible cases:

Either it's a "simple" department with an array of people – then we can sum the salaries in a simple loop.

Or it's an object with N subdepartments – then we can make N recursive calls to get the sum for each of the subdepts and combine the results.

The 1st case is the base of recursion, the trivial case, when we get an array.

The 2nd case when we get an object is the recursive step. A complex task is split into subtasks for smaller departments. They may in turn split again, but sooner or later the split will finish at (1).

The algorithm is probably even easier to read from the code:

```

    let company = { // the same object, compressed for brevity
sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 1600 }],
development: {
  sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],
  internals: [{name: 'Jack', salary: 1300}]
}
};

// The function to do the job
function sumSalaries(department) {
  if (Array.isArray(department)) { // case (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // sum the array
  } else { // case (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // recursively call for subdepartments, sum the results
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 7700

```

The code is short and easy to understand (hopefully?). That's the power of recursion. It also works for any level of subdepartment nesting. Here's the diagram of calls:

We can easily see the principle: for an object {...} subcalls are made, while arrays [...] are the “leaves” of the recursion tree, they give immediate result. Note that the code uses smart features that we've covered before:

Method `arr.reduce` explained in the chapter Array methods to get the sum of the array. Loop `for(val of Object.values(obj))` to iterate over object values: `Object.values` returns an array of them.

Recursive structures A recursive (recursively-defined) data structure is a structure that replicates itself in parts.

We've just seen it in the example of a company structure above.

A company department is:

Either an array of people.

Or an object with departments.

For web-developers there are much better-known examples: HTML and XML documents.

In the HTML document, an HTML-tag may contain a list of:

Text pieces.

HTML-comments.

Other HTML-tags (that in turn may contain text pieces/comments or other tags etc).

That's once again a recursive definition.

For better understanding, we'll cover one more recursive structure named "Linked list" that might be a better alternative for arrays in some cases.

Linked list Imagine, we want to store an ordered list of objects.

The natural choice would be an array:

```
let arr = [obj1, obj2, obj3];
```

...But there's a problem with arrays. The "delete element" and "insert element" operations are expensive. For instance, `arr.unshift(obj)` operation has to renumber all elements to make room for a new obj, and if the array is big, it takes time. Same with `arr.shift()`.

The only structural modifications that do not require mass-renumbering are those that operate with the end of array: `arr.push/pop`. So an array can be quite slow for big queues, when we have to work with the beginning.

Alternatively, if we really need fast insertion/deletion, we can choose another data structure called a linked list.

The linked list element is recursively defined as an object with:

value.

next property referencing the next linked list element or null if that's the end.

For instance:

```
    let list = {  
value: 1,  
next: {  
  value: 2,  
  next: {  
    value: 3,  
    next: {  
      value: 4,  
      next: null  
    }  
  }  
}  
};
```

Graphical representation of the list:

An alternative code for creation:

```
let list = { value: 1 };  
list.next = { value: 2 };  
list.next.next = { value: 3 };  
list.next.next.next = { value: 4 };  
list.next.next.next.next = null;
```

Here we can even more clearly see that there are multiple objects, each one has the value and next pointing to the neighbour. The list variable is the first object in the chain, so following next pointers from it we can reach any element. The list can be easily split into multiple parts and later joined back:

```
    let secondList = list.next.next;  
list.next.next = null;
```

To join:

```
list.next.next = secondList;
```

And surely we can insert or remove items in any place.
For instance, to prepend a new value, we need to update the head of the list:

```
    let list = { value: 1 };  
list.next = { value: 2 };  
list.next.next = { value: 3 };  
list.next.next.next = { value: 4 };  
  
// prepend the new value to the list  
list = { value: "new item", next: list };
```

To remove a value from the middle, change next of the previous one:

```
list.next = list.next.next;
```

We made `list.next` jump over 1 to value 2. The value 1 is now excluded from the chain. If it's not stored anywhere else, it will be automatically removed from the memory. Unlike arrays, there's no mass-renumbering, we can easily rearrange elements. Naturally, lists are not always better than arrays. Otherwise everyone would use only lists.

The main drawback is that we can't easily access an element by its number. In an array that's easy: `arr[n]` is a direct reference. But in the list we need to start from the first item and go next N times to get the Nth element.

...But we don't always need such operations. For instance, when we need a queue or even a deque – the ordered structure that must allow very fast adding/removing elements from both ends, but access to its middle is not needed.

Lists can be enhanced:

We can add property `prev` in addition to `next` to reference the previous element, to move back easily.

We can also add a variable named `tail` referencing the last element of the list (and update it when adding/removing elements from the end).

...The data structure may vary according to our needs.

SummaryTerms:

Recursion is a programming term that means calling a function from itself. Recursive functions can be used to solve tasks in elegant ways.

When a function calls itself, that's called a recursion step. The basis of recursion is function arguments that make the task so simple that the function does not make further calls.

A recursively-defined data structure is a data structure that can be defined using itself. For instance, the linked list can be defined as a data structure consisting of an object referencing a list (or null).

```
list = { value, next -> list }
```

Trees like HTML elements tree or the department tree from this chapter are also naturally recursive: they have branches and every branch can have other branches. Recursive functions can be used to walk them as we've seen in the sumSalary example.

Any recursive function can be rewritten into an iterative one. And that's sometimes required to optimize stuff. But for many tasks a recursive solution is fast enough and easier to write and support.

Task: Sum all numbers till the given one importance: 5 Write a function sumTo(n) that calculates the sum of numbers $1 + 2 + \dots + n$.

For instance:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Make 3 solution variants:

Using a for loop.

Using a recursion, cause $\text{sumTo}(n) = n + \text{sumTo}(n-1)$ for $n > 1$.

Using the arithmetic progression formula.

An example of the result:

```
function sumTo(n) { /*... your code ... */ }

alert( sumTo(100) ); // 5050
```

P.S. Which solution variant is the fastest? The slowest? Why?

P.P.S. Can we use recursion to count sumTo(100000)?

solution The solution using a loop:

```
function sumTo(n) {  
  let sum = 0;  
  for (let i = 1; i <= n; i++) {  
    sum += i;  
  }  
  return sum;  
}  
  
alert( sumTo(100) );
```

The solution using recursion:

```
function sumTo(n) {  
  if (n == 1) return 1;  
  return n + sumTo(n - 1);  
}  
  
alert( sumTo(100) );
```


The solution using the formula: $\text{sumTo}(n) = n \cdot (n+1) / 2$:

```
function sumTo(n) {  
  return n * (n + 1) / 2;  
}
```

```
alert( sumTo(100) );
```

P.S. Naturally, the formula is the fastest solution. It uses only 3 operations for any number n . The math helps!

The loop variant is the second in terms of speed. In both the recursive and the loop variant we sum the same numbers. But the recursion involves nested calls and execution stack management. That also takes resources, so it's slower.

P.P.S. Some engines support the "tail call" optimization: if a recursive call is the very last one in the function, with no other calculations performed, then the outer function will not need to resume the execution, so the engine doesn't need to remember its execution context. That removes the burden on memory. But if the JavaScript engine does not support tail call optimization (most of them don't), there will be an error: maximum stack size exceeded, because there's usually a limitation on the total stack size.

Calculate factorial importance: 4The factorial of a natural number is a number multiplied by "number minus one", then by "number minus two", and so on till 1. The factorial of n is denoted as $n!$

We can write a definition of factorial like this:

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

Values of factorials for different n :

$1! = 1$
 $2! = 2 * 1 = 2$
 $3! = 3 * 2 * 1 = 6$
 $4! = 4 * 3 * 2 * 1 = 24$
 $5! = 5 * 4 * 3 * 2 * 1 = 120$

The task is to write a function `factorial(n)` that calculates $n!$ using recursive calls.

```
alert( factorial(5) ); // 120
```

P.S. Hint: $n!$ can be written as $n * (n-1)!$ For instance: $3! = 3*2! = 3*2*1! = 6$
By definition, a factorial $n!$ can be written as $n * (n-1)!$.
In other words, the result of `factorial(n)` can be calculated as n multiplied by the result of `factorial(n-1)`. And the call for $n-1$ can recursively descend lower, and lower, till 1.

```
function factorial(n) {  
  return (n !== 1) ? n * factorial(n - 1) : 1;  
}
```

```
alert( factorial(5) ); // 120
```

The basis of recursion is the value 1. We can also make 0 the basis here, doesn't matter much, but gives one more recursive step:

```

    function factorial(n) {
    return n ? n * factorial(n - 1) : 1;
    }

    alert( factorial(5) ); // 120

```

Fibonacci numbers importance: 5 The sequence of Fibonacci numbers has the formula $F_n = F_{n-1} + F_{n-2}$. In other words, the next number is a sum of the two preceding ones.

First two numbers are 1, then 2(1+1), then 3(1+2), 5(2+3) and so on: 1, 1, 2, 3, 5, 8, 13, 21....

Fibonacci numbers are related to the Golden ratio and many natural phenomena around us.

Write a function `fib(n)` that returns the n -th Fibonacci number.

An example of work:

```

    function fib(n) { /* your code */ }

    alert(fib(3)); // 2
    alert(fib(7)); // 13
    alert(fib(77)); // 5527939700884757

```

P.S. The function should be fast. The call to `fib(77)` should take no more than a fraction of a second.

solution The first solution we could try here is the recursive one.

Fibonacci numbers are recursive by definition:

```
function fib(n) {
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
```

```
alert( fib(3) ); // 2
alert( fib(7) ); // 13
// fib(77); // will be extremely slow!
```

...But for big values of n it's very slow. For instance, $\text{fib}(77)$ may hang up the engine for some time eating all CPU resources.

That's because the function makes too many subcalls. The same values are re-evaluated again and again.

For instance, let's see a piece of calculations for $\text{fib}(5)$:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

Here we can see that the value of $\text{fib}(3)$ is needed for both $\text{fib}(5)$ and $\text{fib}(4)$. So $\text{fib}(3)$ will be called and evaluated two times completely independently. Here's the full recursion tree:

We can clearly notice that $\text{fib}(3)$ is evaluated two times and $\text{fib}(2)$ is evaluated three

times. The total amount of computations grows much faster than n , making it enormous even for $n=77$.

We can optimize that by remembering already-evaluated values: if a value of say $\text{fib}(3)$ is calculated once, then we can just reuse it in future computations.

Another variant would be to give up recursion and use a totally different loop-based algorithm.

Instead of going from n down to lower values, we can make a loop that starts from 1 and 2, then gets $\text{fib}(3)$ as their sum, then $\text{fib}(4)$ as the sum of two previous values, then $\text{fib}(5)$ and goes up and up, till it gets to the needed value. On each step we only need to remember two previous values.

Here are the steps of the new algorithm in details.

The start:

```
// a = fib(1), b = fib(2), these values are by definition 1
let a = 1, b = 1;
```

```
// get c = fib(3) as their sum
let c = a + b;
```

```
/* we now have fib(1), fib(2), fib(3)
a b c
1, 1, 2
*/
```

Now we want to get $\text{fib}(4) = \text{fib}(2) + \text{fib}(3)$.

Let's shift the variables: a, b will get $\text{fib}(2), \text{fib}(3)$, and c will get their sum:

```
a = b; // now a = fib(2)
b = c; // now b = fib(3)
c = a + b; // c = fib(4)
```

```
/* now we have the sequence:
```

```
  a b c
1, 1, 2, 3
*/
```

The next step gives another sequence number:

```
    a = b; // now a = fib(3)
b = c; // now b = fib(4)
c = a + b; // c = fib(5)

/* now the sequence is (one more number):
   a b c
1, 1, 2, 3, 5
*/
```

...And so on until we get the needed value. That's much faster than recursion and involves no duplicate computations.

The full code:

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757
```

The loop starts with $i=3$, because the first and the second sequence values are hard-coded into variables $a=1$, $b=1$.

The approach is called dynamic programming bottom-up.

Output a single-linked list importance: 5 Let's say we have a single-linked list (as described in the chapter Recursion and stack):

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

Write a function `printList(list)` that outputs list items one-by-one.
Make two variants of the solution: using a loop and using recursion.
What's better: with recursion or without it?

solution Loop-based solution Loop-based solution The loop-based variant of the solution:

```
let list = {
  value: 1,
  next: {
```

```

    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printList(list) {
  let tmp = list;

  while (tmp) {
    alert(tmp.value);
    tmp = tmp.next;
  }
}

printList(list);

```

Please note that we use a temporary variable `tmp` to walk over the list. Technically, we could use a function parameter `list` instead:

```

function printList(list) {

  while(list) {
    alert(list.value);
    list = list.next;
  }

}

```

...But that would be unwise. In the future we may need to extend a function, do something else with the list. If we change `list`, then we lose such ability. Talking about good variable names, `list` here is the list itself. The first element of it. And it should remain like that. That's clear and reliable.

From the other side, the role of tmp is exclusively a list traversal, like i in the for loop.
Recursive solution
Recursive solution
The recursive variant of printList(list) follows a simple logic: to output a list we should output the current element list, then do the same for list.next:

```
    let list = {
value: 1,
next: {
  value: 2,
  next: {
    value: 3,
    next: {
      value: 4,
      next: null
    }
  }
}
};

function printList(list) {

  alert(list.value); // output the current item

  if (list.next) {
    printList(list.next); // do the same for the rest of the list
  }

}

printList(list);
```

Now what's better?
Technically, the loop is more effective. These two variants do the same, but the loop

does not spend resources for nested function calls.

From the other side, the recursive variant is shorter and sometimes easier to understand.

Output a single-linked list in the reverse orderimportance: 5Output a single-linked list from the previous task Output a single-linked list in the reverse order.

Make two solutions: using a loop and using a recursion.

solutionUsing a recursionUsing a recursionThe recursive logic is a little bit tricky here.

We need to first output the rest of the list and then output the current one:

```
    let list = {
value: 1,
next: {
  value: 2,
  next: {
    value: 3,
    next: {
      value: 4,
      next: null
    }
  }
}
};

function printReverseList(list) {

  if (list.next) {
    printReverseList(list.next);
  }

  alert(list.value);
}

printReverseList(list);
```

Using a loopUsing a loopThe loop variant is also a little bit more complicated than the direct output.

There is no way to get the last value in our list. We also can't "go back".

So what we can do is to first go through the items in the direct order and remember them in an array, and then output what we remembered in the reverse order:

```
let list = {  
  value: 1,  
  next: {  
    value: 2,  
    next: {  
      value: 3,  
      next: {  
        value: 4,  
        next: null  
      }  
    }  
  }  
};
```

```
function printReverseList(list) {  
  let arr = [];  
  let tmp = list;  
  
  while (tmp) {  
    arr.push(tmp.value);  
    tmp = tmp.next;  
  }  
  
  for (let i = arr.length - 1; i >= 0; i--) {  
    alert( arr[i] );  
  }  
}
```

```
printReverseList(list);
```

Please note that the recursive solution actually does exactly the same: it follows the list, remembers the items in the chain of nested calls (in the execution context stack), and then outputs them.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/recursion", "identifier": "V
recursion" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Advanced working with
functions", "item": "https://javascript.info/advanced-functions"}]]October 18, 2022Rest
parameters and spread syntaxMany JavaScript built-in functions support an arbitrary
number of arguments.
For instance:

Math.max(arg1, arg2, ..., argN) – returns the greatest of the arguments.

Object.assign(dest, src1, ..., srcN) – copies properties from src1..N into dest.
...and so on.

In this chapter we'll learn how to do the same. And also, how to pass arrays to such functions as parameters.

Rest parameters ...A function can be called with any number of arguments, no matter how it is defined.

Like here:

```
function sum(a, b) {  
  return a + b;  
}  
  
alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted, so the result in the code above is 3.

The rest of the parameters can be included in the function definition by using three dots ... followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

For instance, to gather all arguments into array args:

```
function sumAll(...args) { // args is the name for the array  
  let sum = 0;  
  
  for (let arg of args) sum += arg;  
  
  return sum;  
}  
  
alert( sumAll(1) ); // 1  
alert( sumAll(1, 2) ); // 3  
alert( sumAll(1, 2, 3) ); // 6
```

We can choose to get the first parameters as variables, and gather only the rest. Here the first two arguments go into variables and the rest go into titles array:

```

    function showName(firstName, lastName, ...titles) {
    alert( firstName + ' ' + lastName ); // Julius Caesar

    // the rest go into titles array
    // i.e. titles = ["Consul", "Imperator"]
    alert( titles[0] ); // Consul
    alert( titles[1] ); // Imperator
    alert( titles.length ); // 2
    }

    showName("Julius", "Caesar", "Consul", "Imperator");

```

The rest parameters must be at the end
 The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```

    function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
    // error
    }

```

The ...rest must always be last.

The “arguments” variable There is also a special array-like object named arguments that contains all arguments by their index.
 For instance:

```

    function showName() {
    alert( arguments.length );
    alert( arguments[0] );
    alert( arguments[1] );

    // it's iterable
    // for(let arg of arguments) alert(arg);
    }

    // shows: 2, Julius, Caesar
    showName("Julius", "Caesar");

    // shows: 1, Ilya, undefined (no second argument)
    showName("Ilya");

```

In old times, rest parameters did not exist in the language, and using arguments was the only way to get all arguments of the function. And it still works, we can find it in the old code.

But the downside is that although arguments is both array-like and iterable, it's not an array. It does not support array methods, so we can't call arguments.map(...) for example.

Also, it always contains all arguments. We can't capture them partially, like we did with rest parameters.

So when we need these features, then rest parameters are preferred.

Arrow functions do not have "arguments"

If we access the arguments object from an arrow function, it takes them from the outer "normal" function.

Here's an example:

```
function f() {  
  let showArg = () => alert(arguments[0]);  
  showArg();  
}  
  
f(1); // 1
```

As we remember, arrow functions don't have their own this. Now we know they don't have the special arguments object either.

Spread syntax We've just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there's a built-in function `Math.max` that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let's say we have an array `[3, 5, 1]`. How do we call `Math.max` with it? Passing it "as is" won't work, because `Math.max` expects a list of numeric arguments, not a single array:


```
let arr = [3, 5, 1];  
  
alert( Math.max(arr) ); // NaN
```

And surely we can't manually list items in the code `Math.max(arr[0], arr[1], arr[2])`, because we may be unsure how many there are. As our script executes, there could be a lot, or there could be none. And that would get ugly.
Spread syntax to the rescue! It looks similar to rest parameters, also using `...`, but does quite the opposite.
When `...arr` is used in the function call, it “expands” an iterable object `arr` into the list of arguments.
For `Math.max`:

```
let arr = [3, 5, 1];  
  
alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

We also can pass multiple iterables this way:

```
    let arr1 = [1, -2, 3, 4];  
let arr2 = [8, 3, -8, 1];  
  
alert( Math.max(...arr1, ...arr2) ); // 8
```

We can even combine the spread syntax with normal values:

```
    let arr1 = [1, -2, 3, 4];  
let arr2 = [8, 3, -8, 1];  
  
alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Also, the spread syntax can be used to merge arrays:

```
    let arr = [3, 5, 1];  
let arr2 = [8, 9, 15];
```

```
let merged = [0, ...arr, 2, ...arr2];  
alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

In the examples above we used an array to demonstrate the spread syntax, but any iterable will do.

For instance, here we use the spread syntax to turn the string into array of characters:

```
let str = "Hello";  
alert( [...str] ); // H,e,l,l,o
```

The spread syntax internally uses iterators to gather elements, the same way as `for..of` does.

So, for a string, `for..of` returns characters and `...str` becomes `"H","e","l","l","o"`. The list of characters is passed to array initializer `[...str]`.

For this particular task we could also use `Array.from`, because it converts an iterable (like a string) into an array:

```
let str = "Hello";

// Array.from converts an iterable into an array
alert( Array.from(str) ); // H,e,l,l,o
```

The result is the same as [...str].
But there's a subtle difference between Array.from(obj) and [...obj]:

Array.from operates on both array-likes and iterables.
The spread syntax works only with iterables.

So, for the task of turning something into an array, Array.from tends to be more universal.
Copy an array/object Remember when we talked about Object.assign() in the past?
It is possible to do the same thing with the spread syntax.

```
let arr = [1, 2, 3];

let arrCopy = [...arr]; // spread the array into a list of parameters
                        // then put the result into a new array

// do the arrays have the same contents?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// are the arrays equal?
alert(arr === arrCopy); // false (not same reference)

// modifying our initial array does not modify the copy:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Note that it is possible to do the same thing to make a copy of an object:

```
let obj = { a: 1, b: 2, c: 3 };

let objCopy = { ...obj }; // spread the object into a list of parameters
                        // then return the result in a new object

// do the objects have the same contents?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// are the objects equal?
alert(obj === objCopy); // false (not same reference)

// modifying our initial object does not modify the copy:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

This way of copying an object is much shorter than `let objCopy = Object.assign({}, obj)` or for an array `let arrCopy = Object.assign([], arr)` so we prefer to use it whenever we can.

Summary When we see `"..."` in the code, it is either rest parameters or the spread syntax.

There's an easy way to distinguish between them:

When `...` is at the end of function parameters, it's "rest parameters" and gathers the rest of the list of arguments into an array.

When `...` occurs in a function call or alike, it's called a "spread syntax" and expands an array into a list.

Use patterns:

Rest parameters are used to create functions that accept any number of arguments. The spread syntax is used to pass an array to functions that normally require a list of many arguments.

Together they help to travel between a list and an array of parameters with ease. All arguments of a function call are also available in “old-style” arguments: array-like iterable object.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/rest-parameters-spread", "identifier": "rest-parameters-spread" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Advanced working with functions", "item": "https://javascript.info/advanced-functions"}]}June 13, 2022Variable scope, closureJavaScript is a very function-oriented language. It gives us a lot of freedom. A function can be created at any moment, passed as an argument to another function, and then called from a totally different place of code later.

We already know that a function can access variables outside of it (“outer” variables). But what happens if outer variables change since a function is created? Will the function get newer values or the old ones?

And what if a function is passed along as an argument and called from another place of code, will it get access to outer variables at the new place?

Let’s expand our knowledge to understand these scenarios and more complex ones.

We’ll talk about let/const variables here

In JavaScript, there are 3 ways to declare a variable: let, const (the modern ones), and var (the remnant of the past).

In this article we’ll use let variables in examples.

Variables, declared with const, behave the same, so this article is about const too.

The old var has some notable differences, they will be covered in the article The old “var”.

Code blocksIf a variable is declared inside a code block {...}, it’s only visible inside that block.

For example:

```
{  
  // do some job with local variables that should not be seen outside  
  
  let message = "Hello"; // only visible in this block  
  
  alert(message); // Hello  
}  
  
alert(message); // Error: message is not defined
```

We can use this to isolate a piece of code that does its own task, with variables that only belong to it:

```
{  
  // show message  
  let message = "Hello";  
  alert(message);  
}  
  
{  
  // show another message  
  let message = "Goodbye";
```

```
    alert(message);  
}
```

There'd be an error without blocks

Please note, without separate blocks there would be an error, if we use let with the existing variable name:

```
    // show message  
let message = "Hello";  
alert(message);  
  
// show another message  
let message = "Goodbye"; // Error: variable already declared  
alert(message);
```

For if, for, while and so on, variables declared in {...} are also only visible inside:

```
if (true) {
```



```
let phrase = "Hello!";

alert(phrase); // Hello!
}

alert(phrase); // Error, no such variable!
```

Here, after if finishes, the alert below won't see the phrase, hence the error. That's great, as it allows us to create block-local variables, specific to an if branch. The similar thing holds true for for and while loops:

```
for (let i = 0; i < 3; i++) {
  // the variable i is only visible inside this for
  alert(i); // 0, then 1, then 2
}

alert(i); // Error, no such variable
```

Visually, let i is outside of {...}. But the for construct is special here: the variable, declared inside it, is considered a part of the block. Nested functionsA function is called “nested” when it is created inside another function. It is easily possible to do this with JavaScript. We can use it to organize our code, like this:

```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
```

```
    return firstName + " " + lastName;
}

alert( "Hello, " + getFullName() );
alert( "Bye, " + getFullName() );

}
```

Here the nested function `getFullName()` is made for convenience. It can access the outer variables and so can return the full name. Nested functions are quite common in JavaScript.

What's much more interesting, a nested function can be returned: either as a property of a new object or as a result by itself. It can then be used somewhere else. No matter where, it still has access to the same outer variables.

Below, `makeCounter` creates the “counter” function that returns the next number on each invocation:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Despite being simple, slightly modified variants of that code have practical uses, for instance, as a random number generator to generate random values for automated tests.

How does this work? If we create multiple counters, will they be independent? What's going on with the variables here?

Understanding such things is great for the overall knowledge of JavaScript and beneficial for more complex scenarios. So let's go a bit in-depth.

Lexical Environment

Here be dragons!

The in-depth technical explanation lies ahead.

As far as I'd like to avoid low-level language details, any understanding without them would be lacking and incomplete, so get ready.

For clarity, the explanation is split into multiple steps.

Step 1. Variables In JavaScript, every running function, code block {...}, and the script as a whole have an internal (hidden) associated object known as the Lexical Environment. The Lexical Environment object consists of two parts:

Environment Record – an object that stores all local variables as its properties (and some other information like the value of this).

A reference to the outer lexical environment, the one associated with the outer code.

A “variable” is just a property of the special internal object, Environment Record. “To get or change a variable” means “to get or change a property of that object”.

In this simple code without functions, there is only one Lexical Environment:

This is the so-called global Lexical Environment, associated with the whole script. On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that's why the arrow points to null.

As the code starts executing and goes on, the Lexical Environment changes. Here's a little bit longer code:

Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

When the script starts, the Lexical Environment is pre-populated with all declared variables.

Initially, they are in the “Uninitialized” state. That’s a special internal state, it means that the engine knows about the variable, but it cannot be referenced until it has been declared with `let`. It’s almost the same as if the variable didn’t exist.

Then `let` phrase definition appears. There’s no assignment yet, so its value is undefined. We can use the variable from this point forward.
phrase is assigned a value.
phrase changes the value.

Everything looks simple for now, right?

A variable is a property of a special internal object, associated with the currently executing block/function/script.
Working with variables is actually working with the properties of that object.

Lexical Environment is a specification object
“Lexical Environment” is a specification object: it only exists “theoretically” in the language specification to describe how things work. We can’t get this object in our code and manipulate it directly.
JavaScript engines also may optimize it, discard variables that are unused to save memory and perform other internal tricks, as long as the visible behavior remains as described.

Step 2. Function DeclarationsA function is also a value, like a variable.
The difference is that a Function Declaration is instantly fully initialized.
When a Lexical Environment is created, a Function Declaration immediately becomes a ready-to-use function (unlike `let`, that is unusable till the declaration).
That’s why we can use a function, declared as Function Declaration, even before the declaration itself.
For example, here’s the initial state of the global Lexical Environment when we add a function:

Naturally, this behavior only applies to Function Declarations, not Function Expressions where we assign a function to a variable, such as `let say = function(name)....`

Step 3. Inner and outer Lexical EnvironmentWhen a function runs, at the beginning of the call, a new Lexical Environment is created automatically to store local variables and parameters of the call.
For instance, for `say("John")`, it looks like this (the execution is at the line, labelled with

an arrow):

During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global):

The inner Lexical Environment corresponds to the current execution of say. It has a single property: name, the function argument. We called say("John"), so the value of the name is "John".

The outer Lexical Environment is the global Lexical Environment. It has the phrase variable and the function itself.

The inner Lexical Environment has a reference to the outer one.

When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.

If a variable is not found anywhere, that's an error in strict mode (without use strict, an assignment to a non-existing variable creates a new global variable, for compatibility with old code).

In this example the search proceeds as follows:

For the name variable, the alert inside say finds it immediately in the inner Lexical Environment.

When it wants to access phrase, then there is no phrase locally, so it follows the reference to the outer Lexical Environment and finds it there.

Step 4. Returning a functionLet's return to the makeCounter example.

```
function makeCounter() {  
  let count = 0;  
  
  return function() {  
    return count++;  
  };  
}
```

```
let counter = makeCounter();
```

At the beginning of each `makeCounter()` call, a new Lexical Environment object is created, to store variables for this `makeCounter` run. So we have two nested Lexical Environments, just like in the example above:

What's different is that, during the execution of `makeCounter()`, a tiny nested function is created of only one line: `return count++`. We don't run it yet, only create. All functions remember the Lexical Environment in which they were made. Technically, there's no magic here: all functions have the hidden property named `[[Environment]]`, that keeps the reference to the Lexical Environment where the function was created:

So, `counter. [[Environment]]` has the reference to `{count: 0}` Lexical Environment. That's how the function remembers where it was created, no matter where it's called. The `[[Environment]]` reference is set once and forever at function creation time. Later, when `counter()` is called, a new Lexical Environment is created for the call, and its outer Lexical Environment reference is taken from `counter. [[Environment]]`:

Now when the code inside `counter()` looks for `count` variable, it first searches its own Lexical Environment (empty, as there are no local variables there), then the Lexical Environment of the outer `makeCounter()` call, where it finds and changes it. A variable is updated in the Lexical Environment where it lives. Here's the state after the execution:

If we call `counter()` multiple times, the `count` variable will be increased to 2, 3 and so on, at the same place.

Closure

There is a general programming term “closure”, that developers generally should know.

A closure is a function that remembers its outer variables and can access them. In some languages, that's not possible, or a function should be written in a special way to make it happen. But as explained above, in JavaScript, all functions are naturally closures (there is only one exception, to be covered in The "new Function" syntax).

That is: they automatically remember where they were created using a hidden `[[Environment]]` property, and then their code can access outer variables.

When on an interview, a frontend developer gets a question about “what's a closure?”, a valid answer would be a definition of the closure and an explanation that all functions in JavaScript are closures, and maybe a few more words about technical details: the `[[Environment]]` property and how Lexical Environments work.

Garbage collection Usually, a Lexical Environment is removed from memory with all the variables after the function call finishes. That's because there are no references to it. As any JavaScript object, it's only kept in memory while it's reachable.

However, if there's a nested function that is still reachable after the end of a function, then it has `[[Environment]]` property that references the lexical environment.

In that case the Lexical Environment is still reachable even after the completion of the function, so it stays alive.

For example:

```
function f() {  
  let value = 123;  
  
  return function() {  
    alert(value);  
  }  
}
```

```
let g = f(); // g. [[Environment]] stores a reference to the Lexical Environment  
// of the corresponding f() call
```

Please note that if `f()` is called many times, and resulting functions are saved, then all corresponding Lexical Environment objects will also be retained in memory. In the code below, all 3 of them:

```

    function f() {
    let value = Math.random();

    return function() { alert(value); };
    }

```

```

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
let arr = [f(), f(), f()];

```

A Lexical Environment object dies when it becomes unreachable (just like any other object). In other words, it exists only while there's at least one nested function referencing it.

In the code below, after the nested function is removed, its enclosing Lexical Environment (and hence the value) is cleaned from memory:

```

    function f() {
    let value = 123;

    return function() {
    alert(value);
    }
    }

```

```

let g = f(); // while g function exists, the value stays in memory

```

```

g = null; // ...and now the memory is cleaned up

```

Real-life optimizations As we've seen, in theory while a function is alive, all outer variables are also retained.

But in practice, JavaScript engines try to optimize that. They analyze variable usage and if it's obvious from the code that an outer variable is not used – it is removed.

An important side effect in V8 (Chrome, Edge, Opera) is that such variable will become unavailable in debugging.

Try running the example below in Chrome with the Developer Tools open.

When it pauses, in the console type `alert(value)`.


```

    function f() {
    let value = Math.random();

    function g() {
        debugger; // in console: type alert(value); No such variable!
    }

    return g;
}

let g = f();
g();

```

As you could see – there is no such variable! In theory, it should be accessible, but the engine optimized it out. That may lead to funny (if not such time-consuming) debugging issues. One of them – we can see a same-named outer variable instead of the expected one:

```

    let value = "Surprise!";

    function f() {
        let value = "the closest value";

        function g() {

```

```

    debugger; // in console: type alert(value); Surprise!
  }

  return g;
}

let g = f();
g();

```

This feature of V8 is good to know. If you are debugging with Chrome/Edge/Opera, sooner or later you will meet it.

That is not a bug in the debugger, but rather a special feature of V8. Perhaps it will be changed sometime. You can always check for it by running the examples on this page. TasksDoes a function pickup latest changes?importance: 5The function sayHi uses an external variable name. When the function runs, which value is it going to use?

```

    let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // what will it show: "John" or "Pete"?

```

Such situations are common both in browser and server-side development. A function may be scheduled to execute later than it is created, for instance after a user action or a network request.

So, the question is: does it pick up the latest changes?

solutionThe answer is: Pete.

A function gets outer variables as they are now, it uses the most recent values.

Old variable values are not saved anywhere. When a function wants a variable, it takes the current value from its own Lexical Environment or the outer one.

Which variables are available?importance: 5The function makeWorker below makes

another function and returns it. That new function can be called from somewhere else.

Will it have access to the outer variables from its creation place, or the invocation place, or both?

```

    function makeWorker() {
    let name = "Pete";

    return function() {
        alert(name);
    };
    }

```

```

let name = "John";

```

```

// create a function
let work = makeWorker();

```

```

// call it
work(); // what will it show?

```

Which value it will show? "Pete" or "John"?

solutionThe answer is: Pete.

The work() function in the code below gets name from the place of its origin through the outer lexical environment reference:

So, the result is "Pete" here.

But if there were no let name in makeWorker(), then the search would go outside and take the global variable as we can see from the chain above. In that case the result would be "John".

Are counters independent?importance: 5Here we make two counters: counter and counter2 using the same makeCounter function.

Are they independent? What is the second counter going to show? 0,1 or 2,3 or something else?

```

    function makeCounter() {
    let count = 0;

    return function() {

```

```

    return count++;
  };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ?
alert( counter2() ); // ?

```

solutionThe answer: 0,1.
 Functions counter and counter2 are created by different invocations of makeCounter. So they have independent outer Lexical Environments, each one has its own count. Counter object importance: 5 Here a counter object is made with the help of the constructor function.
 Will it work? What will it show?

```

function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?

```

solutionSurely it will work just fine.
 Both nested functions are created within the same outer Lexical Environment, so they

share access to the same count variable:

```
function Counter() {  
  let count = 0;  
  
  this.up = function() {  
    return ++count;  
  };  
  
  this.down = function() {  
    return --count;  
  };  
}  
  
let counter = new Counter();  
  
alert( counter.up() ); // 1  
alert( counter.up() ); // 2  
alert( counter.down() ); // 1
```

Function importance: 5 Look at the code. What will be the result of the call at the last line?

```

    let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(` ${phrase}, ${user}`);
  }
}

sayHi();

```

solutionThe result is an error.

The function sayHi is declared inside the if, so it only lives inside it. There is no sayHi outside.

Sum with closuresimportance: 4Write function sum that works like this: $\text{sum}(a)(b) = a+b$.

Yes, exactly this way, using double parentheses (not a mistype).

For instance:

```

    sum(1)(2) = 3
sum(5)(-1) = 4

```

solutionFor the second parentheses to work, the first ones must return a function.

Like this:

```

function sum(a) {

```

```

    return function(b) {
      return a + b; // takes "a" from the outer lexical environment
    };
  }

  alert( sum(1)(2) ); // 3
  alert( sum(5)(-1) ); // 4

```

Is variable visible? importance: 4 What will be the result of this code?

```

    let x = 1;

    function func() {
      console.log(x); // ?
    }

    let x = 2;
  }

  func();

```

P.S. There's a pitfall in this task. The solution is not obvious.
 solution The result is: error.
 Try running it:

```

    let x = 1;

    function func() {

```

```

    console.log(x); // ReferenceError: Cannot access 'x' before initialization
    let x = 2;
  }

  func();

```

In this example we can observe the peculiar difference between a “non-existing” and “uninitialized” variable.

As you may have read in the article Variable scope, closure, a variable starts in the “uninitialized” state from the moment when the execution enters a code block (or a function). And it stays uninitialized until the corresponding let statement. In other words, a variable technically exists, but can’t be used before let. The code above demonstrates it.

```

    function func() {
      // the local variable x is known to the engine from the beginning of the function,
      // but "uninitialized" (unusable) until let ("dead zone")
      // hence the error

      console.log(x); // ReferenceError: Cannot access 'x' before initialization

      let x = 2;
    }

```

This zone of temporary unusability of a variable (from the beginning of the code block till let) is sometimes called the “dead zone”.

Filter through function importance: 5 We have a built-in method `arr.filter(f)` for arrays. It filters all elements through the function `f`. If it returns true, then that element is returned in the resulting array.

Make a set of “ready to use” filters:

`inBetween(a, b)` – between `a` and `b` or equal to them (inclusively).
`inArray([...])` – in the given array.

The usage must be like this:

`arr.filter(inBetween(3,6))` – selects only values between 3 and 6.
`arr.filter(inArray([1,2,3]))` – selects only elements matching with one of the members of `[1,2,3]`.

For instance:

```
/* .. your code for inBetween and inArray */  
let arr = [1, 2, 3, 4, 5, 6, 7];  
  
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6  
  
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Open a sandbox with tests.solutionFilter inBetweenFilter inBetween

```
function inBetween(a, b) {  
  return function(x) {  
    return x >= a && x <= b;  
  };  
}  
  
let arr = [1, 2, 3, 4, 5, 6, 7];  
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
```

Filter inArrayFilter inArray

```

    function inArray(arr) {
    return function(x) {
    return arr.includes(x);
    };
    }

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2

```

Open the solution with tests in a sandbox. Sort by field importance: 5 We've got an array of objects to sort:

```

let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];

```

The usual way to do that would be:

```

// by name (Ann, John, Pete)
users.sort((a, b) => a.name > b.name ? 1 : -1);

// by age (Pete, Ann, John)
users.sort((a, b) => a.age > b.age ? 1 : -1);

```

Can we make it even less verbose, like this?

```
users.sort(byField('name'));
users.sort(byField('age'));
```

So, instead of writing a function, just put `byField(fieldName)`.
Write the function `byField` that can be used for that.
Open a sandbox with tests.[solution](#)

```
function byField(fieldName){
  return (a, b) => a[fieldName] > b[fieldName] ? 1 : -1;
}
```

Open the solution with tests in a sandbox.[Army of functions](#)
importance: 5
The following code creates an array of shooters.
Every function is meant to output its number. But something is wrong...

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // create a shooter function,
      alert( i ); // that should show its number
    };
    shooters.push(shooter); // and add it to the array
    i++;
  }

  // ...and return the array of shooters
}
```

```
    return shooters;
}
```

```
let army = makeArmy();
```

```
// all shooters show 10 instead of their numbers 0, 1, 2, 3...
army[0](); // 10 from the shooter number 0
army[1](); // 10 from the shooter number 1
army[2](); // 10 ...and so on.
```

Why do all of the shooters show the same value?

Fix the code so that they work as intended.

Open a sandbox with `tests.solution`. Let's examine what exactly happens inside `makeArmy`, and the solution will become obvious.

It creates an empty array shooters:

```
let shooters = [];
```

Fills it with functions via `shooters.push(function)` in the loop.

Every element is a function, so the resulting array looks like this:

[illegible]

];

The array is returned from the function.

Then, later, the call to any member, e.g. `army[5]()` will get the element `army[5]` from the array (which is a function) and calls it.

Now why do all such functions show the same value, 10?

That's because there's no local variable `i` inside shooter functions. When such a function is called, it takes `i` from its outer lexical environment.

Then, what will be the value of `i`?

If we look at the source:

```
function makeArmy() {  
  ...  
  let i = 0;  
  while (i < 10) {  
    let shooter = function() { // shooter function  
      alert( i ); // should show its number  
    };  
    shooters.push(shooter); // add function to the array  
    i++;  
  }  
  ...  
}
```

We can see that all shooter functions are created in the lexical environment of `makeArmy()` function. But when `army[5]()` is called, `makeArmy` has already finished its job, and the final value of `i` is 10 (while stops at `i=10`).

As the result, all shooter functions get the same value from the outer lexical environment and that is, the last value, `i=10`.

As you can see above, on each iteration of a `while {...}` block, a new lexical environment is created. So, to fix this, we can copy the value of `i` into a variable within the `while {...}` block, like this:

```

    function makeArmy() {
let shooters = [];

let i = 0;
while (i < 10) {
    let j = i;
    let shooter = function() { // shooter function
        alert( j ); // should show its number
    };
    shooters.push(shooter);
    i++;
}

return shooters;
}

let army = makeArmy();

// Now the code works correctly
army[0](); // 0
army[5](); // 5

```

Here `let j = i` declares an “iteration-local” variable `j` and copies `i` into it. Primitives are copied “by value”, so we actually get an independent copy of `i`, belonging to the current loop iteration.

The shooters work correctly, because the value of `i` now lives a little bit closer. Not in `makeArmy()` Lexical Environment, but in the Lexical Environment that corresponds to the current loop iteration:

Such a problem could also be avoided if we used `for` in the beginning, like this:

```
function makeArmy() {  
  let shooters = [];  
  
  for(let i = 0; i < 10; i++) {  
    let shooter = function() { // shooter function  
      alert( i ); // should show its number  
    };  
    shooters.push(shooter);  
  }  
  
  return shooters;  
}  
  
let army = makeArmy();  
  
army[0](); // 0  
army[5](); // 5
```

That's essentially the same, because `for` on each iteration generates a new lexical environment, with its own variable `i`. So `shooter` generated in every iteration references its own `i`, from that very iteration.

Now, as you've put so much effort into reading this, and the final recipe is so simple –

just use for, you may wonder – was it worth that?

Well, if you could easily answer the question, you wouldn't read the solution. So, hopefully this task must have helped you to understand things a bit better.

Besides, there are indeed cases when one prefers while to for, and other scenarios, where such problems are real.

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/vclosure","identifier":"vclosure"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageAdvanced working with functions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Advanced working with functions","item":"https://javascript.info/advanced-functions"}]}November 13, 2022The old "var"

This article is for understanding old scripts

The information in this article is useful for understanding old scripts.

That's not how we write new code.

In the very first chapter about variables, we mentioned three ways of variable declaration:

let
const
var

The var declaration is similar to let. Most of the time we can replace let by var or vice-versa and expect things to work:


```
var message = "Hi";  
alert(message); // Hi
```

But internally `var` is a very different beast, that originates from very old times. It's generally not used in modern scripts, but still lurks in the old ones. If you don't plan on meeting such scripts you may even skip this chapter or postpone it. On the other hand, it's important to understand differences when migrating old scripts from `var` to `let`, to avoid odd errors.

"`var`" has no block scope. Variables, declared with `var`, are either function-scoped or global-scoped. They are visible through blocks. For instance:

```
if (true) {  
  var test = true; // use "var" instead of "let"  
}  
  
alert(test); // true, the variable lives after if
```

As `var` ignores code blocks, we've got a global variable `test`. If we used `let test` instead of `var test`, then the variable would only be visible inside `if`:

```
    if (true) {  
    let test = true; // use "let"  
}
```

```
alert(test); // ReferenceError: test is not defined
```

The same thing for loops: var cannot be block- or loop-local:

```
    for (var i = 0; i < 10; i++) {  
    var one = 1;  
    // ...  
}
```

```
alert(i); // 10, "i" is visible after loop, it's a global variable  
alert(one); // 1, "one" is visible after loop, it's a global variable
```

If a code block is inside a function, then var becomes a function-level variable:

```
function sayHi() {
```

```
if (true) {  
  var phrase = "Hello";  
}  
  
alert(phrase); // works  
}  
  
sayHi();  
alert(phrase); // ReferenceError: phrase is not defined
```

As we can see, `var` pierces through `if`, `for` or other code blocks. That's because a long time ago in JavaScript, blocks had no Lexical Environments, and `var` is a remnant of that.

"`var`" tolerates redeclarations If we declare the same variable with `let` twice in the same scope, that's an error:

```
let user;  
let user; // SyntaxError: 'user' has already been declared
```

With `var`, we can redeclare a variable any number of times. If we use `var` with an already-declared variable, it's just ignored:

```
var user = "Pete";

var user = "John"; // this "var" does nothing (already declared)
// ...it doesn't trigger an error

alert(user); // John
```

“var” variables can be declared below their usevar declarations are processed when the function starts (or script starts for globals). In other words, var variables are defined from the beginning of the function, no matter where the definition is (assuming that the definition is not in the nested function). So this code:

```
function sayHi() {
  phrase = "Hello";

  alert(phrase);

  var phrase;
}
sayHi();
```

...Is technically the same as this (moved var phrase above):

```
function sayHi() {  
  var phrase;  
  
  phrase = "Hello";  
  
  alert(phrase);  
}  
sayHi();
```

...Or even as this (remember, code blocks are ignored):

```
function sayHi() {  
  phrase = "Hello"; // (*)  
  
  if (false) {  
    var phrase;  
  }  
  
  alert(phrase);  
}  
sayHi();
```

People also call such behavior “hoisting” (raising), because all var are “hoisted” (raised) to the top of the function. So in the example above, if (false) branch never executes, but that doesn’t matter. The

var inside it is processed in the beginning of the function, so at the moment of (*) the variable exists.

Declarations are hoisted, but assignments are not.

That's best demonstrated with an example:

```
function sayHi() {  
  alert(phrase);  
  
  var phrase = "Hello";  
}  
  
sayHi();
```

The line `var phrase = "Hello"` has two actions in it:

Variable declaration `var`

Variable assignment `=`.

The declaration is processed at the start of function execution ("hoisted"), but the assignment always works at the place where it appears. So the code works essentially like this:

```

    function sayHi() {
    var phrase; // declaration works at the start...

    alert(phrase); // undefined

    phrase = "Hello"; // ...assignment - when the execution reaches it.
    }

    sayHi();

```

Because all var declarations are processed at the function start, we can reference them at any place. But variables are undefined until the assignments. In both examples above, alert runs without an error, because the variable phrase exists. But its value is not yet assigned, so it shows undefined.

IIFE In the past, as there was only var, and it has no block-level visibility, programmers invented a way to emulate it. What they did was called “immediately-invoked function expressions” (abbreviated as IIFE). That’s not something we should use nowadays, but you can find them in old scripts. An IIFE looks like this:

```

(function() {

    var message = "Hello";

    alert(message); // Hello

})();

```

Here, a Function Expression is created and immediately called. So the code executes right away and has its own private variables. The Function Expression is wrapped with parenthesis (function {...}), because when

JavaScript engine encounters "function" in the main code, it understands it as the start of a Function Declaration. But a Function Declaration must have a name, so this kind of code will give an error:

```
    // Tries to declare and immediately call a function
function() { // <-- SyntaxError: Function statements require a function name

    var message = "Hello";

    alert(message); // Hello

};
```

Even if we say: “okay, let’s add a name”, that won’t work, as JavaScript does not allow Function Declarations to be called immediately:

```
    // syntax error because of parentheses below
function go() {

}; // <-- can't call Function Declaration immediately
```


So, the parentheses around the function is a trick to show JavaScript that the function is created in the context of another expression, and hence it's a Function Expression: it needs no name and can be called immediately.
There exist other ways besides parentheses to tell JavaScript that we mean a Function Expression:

```
// Ways to create IIFE

(function() {
  alert("Parentheses around the function");
})();

(function() {
  alert("Parentheses around the whole thing");
})();

!function() {
  alert("Bitwise NOT operator starts the expression");
})();

+function() {
  alert("Unary plus starts the expression");
}();
```

In all the above cases we declare a Function Expression and run it immediately.
Let's note again: nowadays there's no reason to write such code.
Summary There are two main differences of var compared to let/const:

var variables have no block scope, their visibility is scoped to current function, or global, if declared outside function.
var declarations are processed at function start (script start for globals).

There's one more very minor difference related to the global object, that we'll cover in the next chapter.

These differences make var worse than let most of the time. Block-level variables is such a great thing. That's why let was introduced in the standard long ago, and is now a major way (along with const) to declare a variable.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines –

wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/var", "identifier": "var" });  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;
```

TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"},  
{"@type": "ListItem", "position": 3, "name": "Advanced working with functions", "item": "https://javascript.info/advanced-functions"}]
```

July 1, 2022Global objectThe global object provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.

In a browser it is named window, for Node.js it is global, for other environments it may have another name.

Recently, globalThis was added to the language, as a standardized name for a global object, that should be supported across all environments. It's supported in all major browsers.

We'll use window here, assuming that our environment is a browser. If your script may run in other environments, it's better to use globalThis instead.

All properties of the global object can be accessed directly:

```
    alert("Hello");  
// is the same as  
window.alert("Hello");
```

In a browser, global functions and variables declared with `var` (not `let/const`!) become the property of the global object:

```
var gVar = 5;  
  
alert(window.gVar); // 5 (became a property of the global object)
```

Function declarations have the same effect (statements with function keyword in the main code flow, not function expressions).
Please don't rely on that! This behavior exists for compatibility reasons. Modern scripts use JavaScript modules where such a thing doesn't happen.
If we used `let` instead, such thing wouldn't happen:

```
let gLet = 5;  
  
alert(window.gLet); // undefined (doesn't become a property of the global object)
```

If a value is so important that you'd like to make it available globally, write it directly as a property:

```
        // make current user information global, to let all scripts access it
window.currentUser = {
  name: "John"
};

// somewhere else in code
alert(currentUser.name); // John

// or, if we have a local variable with the name "currentUser"
// get it from window explicitly (safe!)
alert(window.currentUser.name); // John
```

That said, using global variables is generally discouraged. There should be as few global variables as possible. The code design where a function gets “input” variables and produces certain “outcome” is clearer, less prone to errors and easier to test than if it uses outer or global variables.

Using for polyfills We use the global object to test for support of modern language features.

For instance, test if a built-in Promise object exists (it doesn't in really old browsers):

```
if (!window.Promise) {
```

```
    alert("Your browser is really old!");  
}
```

If there's none (say, we're in an old browser), we can create “polyfills”: add functions that are not supported by the environment, but exist in the modern standard.

```
    if (!window.Promise) {  
    window.Promise = ... // custom implementation of the modern language feature  
}
```

Summary

The global object holds variables that should be available everywhere.
That includes JavaScript built-ins, such as Array and environment-specific values, such as window.innerHeight – the window height in the browser.

The global object has a universal name globalThis.
...But more often is referred by “old-school” environment-specific names, such as window (browser) and global (Node.js).

We should store values in the global object only if they're truly global for our project.
And keep their number at minimum.

In-browser, unless we're using modules, global functions and variables declared with var become a property of the global object.

To make our code future-proof and easier to understand, we should access properties

of the global object directly, as window.x.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://javascript.info/global-object","identifier":"global-object"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageAdvanced working with functions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Advanced working with functions","item":"https://javascript.info/advanced-functions"}]May 3, 2022Function object, NFEAs we already know, a function in JavaScript is a value. Every value in JavaScript has a type. What type is a function? In JavaScript, functions are objects. A good way to imagine functions is as callable “action objects”. We can not only call them, but also treat them as objects: add/remove properties, pass by reference etc. The “name” propertyFunction objects contain some useable properties. For instance, a function’s name is accessible as the “name” property:

```
function sayHi() {  
  alert("Hi");  
}  
  
alert(sayHi.name); // sayHi
```

What's kind of funny, the name-assigning logic is smart. It also assigns the correct name to a function even if it's created without one, and then immediately assigned:

```
    let sayHi = function() {  
      alert("Hi");  
    };  
  
    alert(sayHi.name); // sayHi (there's a name!)
```

It also works if the assignment is done via a default value:

```
    function f(sayHi = function() {}) {  
      alert(sayHi.name); // sayHi (works!)  
    }  
  
    f();
```

In the specification, this feature is called a “contextual name”. If the function does not provide one, then in an assignment it is figured out from the context. Object methods have names too:

```

    let user = {

sayHi() {
    // ...
},

sayBye: function() {
    // ...
}

}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye

```

There's no magic though. There are cases when there's no way to figure out the right name. In that case, the name property is empty, like here:

```

    // function created inside array
let arr = [function() {}];

alert( arr[0].name ); // <empty string>
// the engine has no way to set up the right name, so there is none

```


In practice, however, most functions do have a name.
The “length” property There is another built-in property “length” that returns the number of function parameters, for instance:

```
function f1(a) {}  
function f2(a, b) {}  
function many(a, b, ...more) {}  
  
alert(f1.length); // 1  
alert(f2.length); // 2  
alert(many.length); // 2
```

Here we can see that rest parameters are not counted.
The length property is sometimes used for introspection in functions that operate on other functions.
For instance, in the code below the ask function accepts a question to ask and an arbitrary number of handler functions to call.
Once a user provides their answer, the function calls the handlers. We can pass two kinds of handlers:

A zero-argument function, which is only called when the user gives a positive answer.
A function with arguments, which is called in either case and returns an answer.

To call handler the right way, we examine the handler.length property.
The idea is that we have a simple, no-arguments handler syntax for positive cases (most frequent variant), but are able to support universal handlers as well:

```

    function ask(question, ...handlers) {
    let isYes = confirm(question);

    for(let handler of handlers) {
      if (handler.length == 0) {
        if (isYes) handler();
      } else {
        handler(isYes);
      }
    }
  }

  // for positive answer, both handlers are called
  // for negative answer, only the second one
  ask("Question?", () => alert('You said yes'), result => alert(result));

```

This is a particular case of so-called polymorphism – treating arguments differently depending on their type or, in our case depending on the length. The idea does have a use in JavaScript libraries.

Custom properties We can also add properties of our own.

Here we add the counter property to track the total calls count:

```

function sayHi() {
  alert("Hi");

```

```

    // let's count how many times we run
    sayHi.counter++;
  }
  sayHi.counter = 0; // initial value

  sayHi(); // Hi
  sayHi(); // Hi

  alert( `Called ${sayHi.counter} times` ); // Called 2 times

```

A property is not a variable

A property assigned to a function like `sayHi.counter = 0` does not define a local variable `counter` inside it. In other words, a property `counter` and a variable `let counter` are two unrelated things.

We can treat a function as an object, store properties in it, but that has no effect on its execution. Variables are not function properties and vice versa. These are just parallel worlds.

Function properties can replace closures sometimes. For instance, we can rewrite the counter function example from the chapter Variable scope, closure to use a function property:

```

    function makeCounter() {
    // instead of:
    // let count = 0

    function counter() {
      return counter.count++;
    };

    counter.count = 0;

```

```
    return counter;
}
```

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

The count is now stored in the function directly, not in its outer Lexical Environment. Is it better or worse than using a closure? The main difference is that if the value of count lives in an outer variable, then external code is unable to access it. Only nested functions may modify it. And if it's bound to a function, then such a thing is possible:

```
function makeCounter() {
    function counter() {
        return counter.count++;
    };

    counter.count = 0;

    return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

So the choice of implementation depends on our aims.

Named Function ExpressionNamed Function Expression, or NFE, is a term for Function Expressions that have a name.
For instance, let's take an ordinary Function Expression:

```
let sayHi = function(who) {  
  alert(`Hello, ${who}`);  
};
```

And add a name to it:

```
let sayHi = function func(who) {  
  alert(`Hello, ${who}`);  
};
```

Did we achieve anything here? What's the purpose of that additional "func" name? First let's note, that we still have a Function Expression. Adding the name "func" after function did not make it a Function Declaration, because it is still created as a part of an assignment expression.
Adding such a name also did not break anything.
The function is still available as sayHi():

```
let sayHi = function func(who) {  
  alert(`Hello, ${who}`);  
};
```

```
sayHi("John"); // Hello, John
```

There are two special things about the name `func`, that are the reasons for it:

It allows the function to reference itself internally.
It is not visible outside of the function.

For instance, the function `sayHi` below calls itself again with "Guest" if no `who` is provided:

```
let sayHi = function func(who) {  
  if (who) {  
    alert(`Hello, ${who}`);  
  } else {  
    func("Guest"); // use func to re-call itself  
  }  
};
```

```
sayHi(); // Hello, Guest
```

// But this won't work:

```
func(); // Error, func is not defined (not visible outside of the function)
```

Why do we use `func`? Maybe just use `sayHi` for the nested call?
Actually, in most cases we can:

```
let sayHi = function(who) {  
  if (who) {
```

```
    alert(`Hello, ${who}`);  
  } else {  
    sayHi("Guest");  
  }  
};
```

The problem with that code is that `sayHi` may change in the outer code. If the function gets assigned to another variable instead, the code will start to give errors:

```
    let sayHi = function(who) {  
      if (who) {  
        alert(`Hello, ${who}`);  
      } else {  
        sayHi("Guest"); // Error: sayHi is not a function  
      }  
    };  
  
    let welcome = sayHi;  
    sayHi = null;  
  
    welcome(); // Error, the nested sayHi call doesn't work any more!
```

That happens because the function takes `sayHi` from its outer lexical environment. There's no local `sayHi`, so the outer variable is used. And at the moment of the call that outer `sayHi` is null.

The optional name which we can put into the Function Expression is meant to solve exactly these kinds of problems.

Let's use it to fix our code:

```

    let sayHi = function func(who) {
    if (who) {
      alert(`Hello, ${who}`);
    } else {
      func("Guest"); // Now all fine
    }
  };

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (nested call works)

```

Now it works, because the name "func" is function-local. It is not taken from outside (and not visible there). The specification guarantees that it will always reference the current function.

The outer code still has its variable sayHi or welcome. And func is an “internal function name”, the way for the function to can call itself reliably.

There’s no such thing for Function Declaration

The “internal name” feature described here is only available for Function Expressions, not for Function Declarations. For Function Declarations, there is no syntax for adding an “internal” name.

Sometimes, when we need a reliable internal name, it’s the reason to rewrite a Function Declaration to Named Function Expression form.

Summary Functions are objects.

Here we covered their properties:

name – the function name. Usually taken from the function definition, but if there’s none, JavaScript tries to guess it from the context (e.g. an assignment).

length – the number of arguments in the function definition. Rest parameters are not counted.

If the function is declared as a Function Expression (not in the main code flow), and it

carries the name, then it is called a Named Function Expression. The name can be used inside to reference itself, for recursive calls or such.

Also, functions may carry additional properties. Many well-known JavaScript libraries make great use of this feature.

They create a “main” function and attach many other “helper” functions to it. For instance, the jQuery library creates a function named \$. The lodash library creates a function `_`, and then adds `_.clone`, `_.keyBy` and other properties to it (see the docs when you want to learn more about them). Actually, they do it to lessen their pollution of the global space, so that a single library gives only one global variable. That reduces the possibility of naming conflicts.

So, a function can do a useful job by itself and also carry a bunch of other functionality in properties.

TasksSet and decrease for counterimportance: 5Modify the code of `makeCounter()` so that the counter can also decrease and set the number:

`counter()` should return the next number (as before).

`counter.set(value)` should set the counter to value.

`counter.decrease()` should decrease the counter by 1.

See the sandbox code for the complete usage example.

P.S. You can use either a closure or the function property to keep the current count. Or write both variants.

Open a sandbox with tests.solutionThe solution uses `count` in the local variable, but addition methods are written right into the counter. They share the same outer lexical environment and also can access the current count.

```
function makeCounter() {
  let count = 0;

  function counter() {
    return count++;
  }

  counter.set = value => count = value;

  counter.decrease = () => count--;

  return counter;
}
```

Open the solution with tests in a sandbox.Sum with an arbitrary amount of

bracketsimportance: 2Write function sum that would work like this:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Hint: you may need to setup custom object to primitive conversion for your function.

Open a sandbox with tests.solution

For the whole thing to work anyhow, the result of sum must be function.

That function must keep in memory the current value between calls.

According to the task, the function must become the number when used in ==.

Functions are objects, so the conversion happens as described in the chapter Object to primitive conversion, and we can provide our own method that returns the number.

Now the code:

```
function sum(a) {

  let currentSum = a;

  function f(b) {
    currentSum += b;
    return f;
  }

  f.toString = function() {
    return currentSum;
  }
}
```

```

};

return f;
}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1)(2) ); // 6
alert( sum(6)(-1)(-2)(-3) ); // 0
alert( sum(0)(1)(2)(3)(4)(5) ); // 15

```

Please note that the sum function actually works only once. It returns function f. Then, on each subsequent call, f adds its parameter to the sum currentSum, and returns itself. There is no recursion in the last line of f. Here is what recursion looks like:

```

function f(b) {
  currentSum += b;
  return f(); // <-- recursive call
}

```

And in our case, we just return the function, without calling it:

```

function f(b) {
  currentSum += b;
  return f; // <-- does not call itself, returns itself
}

```

This f will be used in the next call, again return itself, as many times as needed. Then, when used as a number or a string – the toString returns the currentSum. We could also use Symbol.toPrimitive or valueOf here for the conversion. Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of

commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/function-object","identifier":"function-object"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageAdvanced working with functions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Advanced working with functions","item":"https://javascript.info/advanced-functions"}]}

October 22, 2020The "new Function" syntaxThere's one more way to create a function. It's rarely used, but sometimes there's no alternative.

SyntaxThe syntax for creating a function:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

The function is created with the arguments `arg1...argN` and the given `functionBody`. It's easier to understand by looking at an example. Here's a function with two arguments:

```
let sum = new Function('a', 'b', 'return a + b');

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

And here there's a function without arguments, with only the function body:

```
let sayHi = new Function('alert("Hello")');  
  
sayHi(); // Hello
```

The major difference from other ways we've seen is that the function is created literally from a string, that is passed at run time.
All previous declarations required us, programmers, to write the function code in the script.
But `new Function` allows to turn any string into a function. For example, we can receive a new function from a server and then execute it:

```
let str = ... receive the code from a server dynamically ...  
  
let func = new Function(str);  
func();
```

It is used in very specific cases, like when we receive code from a server, or to dynamically compile a function from a template, in complex web-applications.
Closure Usually, a function remembers where it was born in the special property `[[Environment]]`. It references the Lexical Environment from where it's created (we covered that in the chapter Variable scope, closure).
But when a function is created using `new Function`, its `[[Environment]]` is set to reference not the current Lexical Environment, but the global one.
So, such function doesn't have access to outer variables, only to the global ones.

```
function getFunc() {  
  let value = "test";  
  
  let func = new Function('alert(value)');  
  
  return func;  
}  
  
getFunc(); // error: value is not defined
```

Compare it with the regular behavior:

```
function getFunc() {  
  let value = "test";  
  
  let func = function() { alert(value); };  
  
  return func;  
}  
  
getFunc(); // "test", from the Lexical Environment of getFunc
```

This special feature of new Function looks strange, but appears very useful in practice.

Imagine that we must create a function from a string. The code of that function is not known at the time of writing the script (that's why we don't use regular functions), but will be known in the process of execution. We may receive it from the server or from another source.

Our new function needs to interact with the main script.

What if it could access the outer variables?

The problem is that before JavaScript is published to production, it's compressed using a minifier – a special program that shrinks code by removing extra comments, spaces and – what's important, renames local variables into shorter ones.

For instance, if a function has `let userName`, minifier replaces it with `let a` (or another letter if this one is occupied), and does it everywhere. That's usually a safe thing to do, because the variable is local, nothing outside the function can access it. And inside the function, minifier replaces every mention of it. Minifiers are smart, they analyze the code structure, so they don't break anything. They're not just a dumb find-and-replace. So if `new Function` had access to outer variables, it would be unable to find renamed `userName`.

If `new Function` had access to outer variables, it would have problems with minifiers.

Besides, such code would be architecturally bad and prone to errors.

To pass something to a function, created as `new Function`, we should use its arguments.

SummaryThe syntax:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

For historical reasons, arguments can also be given as a comma-separated list. These three declarations mean the same:

```
new Function('a', 'b', 'return a + b'); // basic syntax
new Function('a,b', 'return a + b'); // comma-separated
new Function('a , b', 'return a + b'); // comma-separated with spaces
```

Functions created with `new Function`, have `[[Environment]]` referencing the global Lexical Environment, not the outer one. Hence, they cannot use outer variables. But that's actually good, because it insures us from errors. Passing parameters explicitly is a much better method architecturally and causes no problems with minifiers.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue

or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...).
`var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/new-function", "identifier": "new-function" }); };
var DisqusShortname = "javascriptinfo";
var DisqusEnabled = true;
TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Advanced working with functions", "item": "https://javascript.info/advanced-functions"}]}
October 3, 2022
Scheduling: setTimeout and setInterval
We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call".
There are two methods for it:`

`setTimeout` allows us to run a function once after the interval of time.
`setInterval` allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

`setTimeout` The syntax:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Parameters:

`func|code`

Function or a string of code to execute.

Usually, that's a function. For historical reasons, a string of code can be passed, but that's not recommended.

`delay`

The delay before run, in milliseconds (1000 ms = 1 second), by default 0.

`arg1, arg2...`

Arguments for the function

For instance, this code calls `sayHi()` after one second:


```
function sayHi() {  
  alert('Hello');  
}  
  
setTimeout(sayHi, 1000);
```

With arguments:

```
function sayHi(phrase, who) {  
  alert( phrase + ', ' + who );  
}  
  
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

If the first argument is a string, then JavaScript creates a function from it.
So, this will also work:

```
setTimeout("alert('Hello')", 1000);
```

But using strings is not recommended, use arrow functions instead of them, like this:

```
setTimeout(() => alert('Hello'), 1000);
```

Pass a function, but don't run it

Novice developers sometimes make a mistake by adding brackets () after the function:

```
// wrong!  
setTimeout(sayHi(), 1000);
```

That doesn't work, because `setTimeout` expects a reference to a function. And here `sayHi()` runs the function, and the result of its execution is passed to `setTimeout`. In our case the result of `sayHi()` is undefined (the function returns nothing), so nothing is scheduled.

Canceling with clearTimeoutA call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

As we can see from alert output, in a browser the timer identifier is a number. In other environments, this can be something else. For instance, Node.js returns a timer object with additional methods.

Again, there is no universal specification for these methods, so that's fine.

For browsers, timers are described in the timers section of HTML Living Standard.

setIntervalThe setInterval method has the same syntax as setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike `setTimeout` it runs the function not only once, but regularly after the given interval of time. To stop further calls, we should call `clearInterval(timerId)`. The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Time goes on while alert is shown
In most browsers, including Chrome and Firefox the internal timer continues “ticking” while showing alert/confirm/prompt. So if you run the code above and don’t dismiss the alert window for some time, then the next alert will be shown immediately as you do it. The actual interval between alerts will be shorter than 2 seconds.

Nested `setTimeout` There are two ways of running something regularly. One is `setInterval`. The other one is a nested `setTimeout`, like this:

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
```

```
    alert('tick');
    timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

The `setTimeout` above schedules the next call right at the end of the current one (*). The nested `setTimeout` is a more flexible method than `setInterval`. This way the next call may be scheduled differently, depending on the results of the current one.

For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds...

Here's the pseudocode:

```
    let delay = 5000;

let timerId = setTimeout(function request() {
    ...send request...

    if (request failed due to server overload) {
        // increase the interval to the next run
        delay *= 2;
    }

    timerId = setTimeout(request, delay);
}, delay);
```

And if the functions that we're scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.

Nested `setTimeout` allows to set the delay between the executions more precisely than `setInterval`.

Let's compare two code fragments. The first one uses `setInterval`:

```
    let i = 1;
setInterval(function() {
    func(i++);
}, 100);
```

The second one uses nested `setTimeout`:

```
    let i = 1;
    setTimeout(function run() {
      func(i++);
      setTimeout(run, 100);
    }, 100);
```

For `setInterval` the internal scheduler will run `func(i++)` every 100ms:

Did you notice?

The real delay between `func` calls for `setInterval` is less than in the code!

That's normal, because the time taken by `func`'s execution "consumes" a part of the interval.

It is possible that `func`'s execution turns out to be longer than we expected and takes more than 100ms.

In this case the engine waits for `func` to complete, then checks the scheduler and if the time is up, runs it again immediately.

In the edge case, if the function always executes longer than delay ms, then the calls will happen without a pause at all.

And here is the picture for the nested `setTimeout`:

The nested `setTimeout` guarantees the fixed delay (here 100ms).

That's because a new call is planned at the end of the previous one.

Garbage collection and `setInterval/setTimeout` callback

When a function is passed in `setInterval/setTimeout`, an internal reference is created to it and saved in the scheduler. It prevents the function from being garbage collected, even if there are no other references to it.

```
// the function stays in memory until the scheduler calls it
setTimeout(function() {...}, 100);
```

For `setInterval` the function stays in memory until `clearInterval` is called. There's a side effect. A function references the outer lexical environment, so, while it lives, outer variables live too. They may take much more memory than the function itself. So when we don't need the scheduled function anymore, it's better to cancel it, even if it's very small.

Zero delay `setTimeout` There's a special use case: `setTimeout(func, 0)`, or just `setTimeout(func)`.

This schedules the execution of `func` as soon as possible. But the scheduler will invoke it only after the currently executing script is complete.

So the function is scheduled to run "right after" the current script.

For instance, this outputs "Hello", then immediately "World":

```
setTimeout(() => alert("World"));

alert("Hello");
```

The first line "puts the call into calendar after 0ms". But the scheduler will only "check the calendar" after the current script is complete, so "Hello" is first, and "World" – after it.

There are also advanced browser-related use cases of zero-delay timeout, that we'll discuss in the chapter Event loop: microtasks and macrotasks.

Zero delay is in fact not zero (in a browser)

In the browser, there's a limitation of how often nested timers can run. The

HTML Living Standard says: “after five nested timers, the interval is forced to be at least 4 milliseconds.”.

Let's demonstrate what it means with the example below. The `setTimeout` call in it re-schedules itself with zero delay. Each call remembers the real time from the previous one in the `times` array. What do the real delays look like? Let's see:

```
    let start = Date.now();
let times = [];

setTimeout(function run() {
    times.push(Date.now() - start); // remember delay from the previous call

    if (start + 100 < Date.now()) alert(times); // show the delays after 100ms
    else setTimeout(run); // else re-schedule
});

// an example of the output:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

First timers run immediately (just as written in the spec), and then we see 9, 15, 20, 24.... The 4+ ms obligatory delay between invocations comes into play.

The similar thing happens if we use `setInterval` instead of `setTimeout`: `setInterval(f)` runs `f` few times with zero-delay, and afterwards with 4+ ms delay.

That limitation comes from ancient times and many scripts rely on it, so it exists for historical reasons.

For server-side JavaScript, that limitation does not exist, and there exist other ways to schedule an immediate asynchronous job, like `setImmediate` for Node.js. So this note is browser-specific.

Summary

Methods `setTimeout(func, delay, ...args)` and `setInterval(func, delay, ...args)` allow us to run the `func` once/regularly after `delay` milliseconds.

To cancel the execution, we should call `clearTimeout/clearInterval` with the value

returned by setTimeout/setInterval.

Nested setTimeout calls are a more flexible alternative to setInterval, allowing us to set the time between executions more precisely.

Zero delay scheduling with setTimeout(func, 0) (the same as setTimeout(func)) is used to schedule the call “as soon as possible, but after the current script is complete”.

The browser limits the minimal delay for five or more nested calls of setTimeout or for setInterval (after 5th call) to 4ms. That’s for historical reasons.

Please note that all scheduling methods do not guarantee the exact delay.

For example, the in-browser timer may slow down for a lot of reasons:

The CPU is overloaded.

The browser tab is in the background mode.

The laptop is on battery saving mode.

All that may increase the minimal timer resolution (the minimal delay) to 300ms or even 1000ms depending on the browser and OS-level performance settings.

Task: Write a function printNumbers(from, to) that outputs a number every second, starting from from and ending with to.

Make two variants of the solution.

Using setInterval.

Using nested setTimeout.

Solution Using setInterval:

```
function printNumbers(from, to) {  
  let current = from;  
  
  let timerId = setInterval(function() {  
    alert(current);  
    if (current == to) {  
      clearInterval(timerId);  
    }  
    current++;  
  }, 1000);  
}
```

```
    }, 1000);  
}
```

```
// usage:  
printNumbers(5, 10);
```

Using nested setTimeout:

```
function printNumbers(from, to) {  
  let current = from;  
  
  setTimeout(function go() {  
    alert(current);  
    if (current < to) {  
      setTimeout(go, 1000);  
    }  
    current++;  
  }, 1000);  
}
```

```
// usage:  
printNumbers(5, 10);
```

Note that in both solutions, there is an initial delay before the first output. The function is called after 1000ms the first time. If we also want the function to run immediately, then we can add an additional call on a separate line, like this:

```

    function printNumbers(from, to) {
    let current = from;

    function go() {
      alert(current);
      if (current == to) {
        clearInterval(timerId);
      }
      current++;
    }

    go();
    let timerId = setInterval(go, 1000);
  }

  printNumbers(5, 10);

```

What will setTimeout show?importance: 5In the code below there's a setTimeout call scheduled, then a heavy calculation is run, that takes more than 100ms to finish. When will the scheduled function run?

After the loop.

Before the loop.

In the beginning of the loop.

What is alert going to show?

```

    let i = 0;

    setTimeout(() => alert(i), 100); // ?

    // assume that the time to execute this function is >100ms
    for(let j = 0; j < 100000000; j++) {

```

```

    i++;
}

```

Any setTimeout will run only after the current code has finished.
The i will be the last one: 100000000.

```

let i = 0;

setTimeout(() => alert(i), 100); // 100000000

// assume that the time to execute this function is >100ms
for(let j = 0; j < 100000000; j++) {
    i++;
}

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/settimeout-setinterval", "identifier": "settimeout-setinterval" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Advanced working with functions", "item": "https://javascript.info/advanced-functions"}]}June 8, 2022Decorators and forwarding, call/applyJavaScript gives exceptional flexibility when dealing with

functions. They can be passed around, used as objects, and now we'll see how to forward calls between them and decorate them.

Transparent cachingLet's say we have a function `slow(x)` which is CPU-heavy, but its results are stable. In other words, for the same `x` it always returns the same result. If the function is called often, we may want to cache (remember) the results to avoid spending extra-time on recalculations.

But instead of adding that functionality into `slow()` we'll create a wrapper function, that adds caching. As we'll see, there are many benefits of doing so.

Here's the code, and explanations follow:

```
function slow(x) {
  // there can be a heavy CPU-intensive job here
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) { // if there's such key in cache
      return cache.get(x); // read the result from it
    }

    let result = func(x); // otherwise call func

    cache.set(x, result); // and cache (remember) the result
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) is cached and the result returned
alert( "Again: " + slow(1) ); // slow(1) result returned from cache
```

```
alert( slow(2) ); // slow(2) is cached and the result returned  
alert( "Again: " + slow(2) ); // slow(2) result returned from cache
```

In the code above `cachingDecorator` is a decorator: a special function that takes another function and alters its behavior.

The idea is that we can call `cachingDecorator` for any function, and it will return the caching wrapper. That's great, because we can have many functions that could use such a feature, and all we need to do is to apply `cachingDecorator` to them.

By separating caching from the main function code we also keep the main code simpler. The result of `cachingDecorator(func)` is a “wrapper”: `function(x)` that “wraps” the call of `func(x)` into caching logic:

From an outside code, the wrapped `slow` function still does the same. It just got a caching aspect added to its behavior.

To summarize, there are several benefits of using a separate `cachingDecorator` instead of altering the code of `slow` itself:

The `cachingDecorator` is reusable. We can apply it to another function.

The caching logic is separate, it did not increase the complexity of `slow` itself (if there was any).

We can combine multiple decorators if needed (other decorators will follow).

Using “`func.call`” for the contextThe caching decorator mentioned above is not suited to work with object methods.

For instance, in the code below `worker.slow()` stops working after the decoration:

```
// we'll make worker.slow caching  
let worker = {
```

```

someMethod() {
  return 1;
},

slow(x) {
  // scary CPU-heavy task here
  alert("Called with " + x);
  return x * this.someMethod(); // (*)
}
};

```

```

// same code as before
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

```

```

alert( worker.slow(1) ); // the original method works

```

```

worker.slow = cachingDecorator(worker.slow); // now make it caching

```

```

alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod' of
undefined

```

The error occurs in the line (*) that tries to access `this.someMethod` and fails. Can you see why?

The reason is that the wrapper calls the original function as `func(x)` in the line (**). And, when called like that, the function gets `this = undefined`.

We would observe a similar symptom if we tried to run:

```

let func = worker.slow;
func(2);

```

So, the wrapper passes the call to the original method, but without the context this. Hence the error.

Let's fix it.

There's a special built-in function method `func.call(context, ...args)` that allows to call a function explicitly setting this.

The syntax is:

```
func.call(context, arg1, arg2, ...)
```

It runs `func` providing the first argument as `this`, and the next as the arguments. To put it simply, these two calls do almost the same:

```
func(1, 2, 3);  
func.call(obj, 1, 2, 3)
```

They both call `func` with arguments 1, 2 and 3. The only difference is that `func.call` also sets `this` to `obj`.

As an example, in the code below we call `sayHi` in the context of different objects: `sayHi.call(user)` runs `sayHi` providing `this=user`, and the next line sets `this=admin`:

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



```
let user = { name: "John" };  
let admin = { name: "Admin" };  
  
// use call to pass different objects as "this"  
sayHi.call( user ); // John  
sayHi.call( admin ); // Admin
```

And here we use call to call say with the given context and phrase:

```
function say(phrase) {  
  alert(this.name + ': ' + phrase);  
}  
  
let user = { name: "John" };  
  
// user becomes this, and "Hello" becomes the first argument  
say.call( user, "Hello" ); // John: Hello
```

In our case, we can use call in the wrapper to pass the context to the original function:

```

    let worker = {
    someMethod() {
        return 1;
    },

    slow(x) {
        alert("Called with " + x);
        return x * this.someMethod(); // (*)
    }
};

function cachingDecorator(func) {
    let cache = new Map();
    return function(x) {
        if (cache.has(x)) {
            return cache.get(x);
        }
        let result = func.call(this, x); // "this" is passed correctly now
        cache.set(x, result);
        return result;
    };
}

```

`worker.slow = cachingDecorator(worker.slow); // now make it caching`

```

alert( worker.slow(2) ); // works
alert( worker.slow(2) ); // works, doesn't call the original (cached)

```

Now everything is fine.

To make it all clear, let's see more deeply how this is passed along:

After the decoration `worker.slow` is now the wrapper function `(x) { ... }`.

So when `worker.slow(2)` is executed, the wrapper gets 2 as an argument and `this=worker` (it's the object before dot).

Inside the wrapper, assuming the result is not yet cached, `func.call(this, x)` passes the current `this` (`=worker`) and the current argument (`=2`) to the original method.

Going multi-argument Now let's make `cachingDecorator` even more universal. Till now it was working only with single-argument functions.

Now how to cache the multi-argument `worker.slow` method?

```

    let worker = {
    slow(min, max) {
      return min + max; // scary CPU-hogger is assumed
    }
  };

  // should remember same-argument calls
  worker.slow = cachingDecorator(worker.slow);

```

Previously, for a single argument x we could just `cache.set(x, result)` to save the result and `cache.get(x)` to retrieve it. But now we need to remember the result for a combination of arguments (min,max). The native Map takes single value only as the key. There are many solutions possible:

Implement a new (or use a third-party) map-like data structure that is more versatile and allows multi-keys.

Use nested maps: `cache.set(min)` will be a Map that stores the pair (max, result). So we can get result as `cache.get(min).get(max)`.

Join two values into one. In our particular case we can just use a string "min,max" as the Map key. For flexibility, we can allow to provide a hashing function for the decorator, that knows how to make one value from many.

For many practical applications, the 3rd variant is good enough, so we'll stick to it. Also we need to pass not just x , but all arguments in `func.call`. Let's recall that in a `function()` we can get a pseudo-array of its arguments as `arguments`, so `func.call(this, x)` should be replaced with `func.call(this, ...arguments)`. Here's a more powerful `cachingDecorator`:

```

    let worker = {
    slow(min, max) {
      alert(`Called with ${min},${max}`);
      return min + max;
    }
  };

```

```

    }
};

function cachingDecorator(func, hash) {
    let cache = new Map();
    return function() {
        let key = hash(arguments); // (*)
        if (cache.has(key)) {
            return cache.get(key);
        }

        let result = func.call(this, ...arguments); // (**)

        cache.set(key, result);
        return result;
    };
}

function hash(args) {
    return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // works
alert( "Again " + worker.slow(3, 5) ); // same (cached)

```

Now it works with any number of arguments (though the hash function would also need to be adjusted to allow any number of arguments. An interesting way to handle this will be covered below).

There are two changes:

In the line (*) it calls hash to create a single key from arguments. Here we use a simple “joining” function that turns arguments (3, 5) into the key "3,5". More complex cases may require other hashing functions.

Then (**) uses func.call(this, ...arguments) to pass both the context and all arguments the wrapper got (not just the first one) to the original function.

func.applyInstead of func.call(this, ...arguments) we could use func.apply(this, arguments).

The syntax of built-in method func.apply is:

```
func.apply(context, args)
```

It runs the func setting this=context and using an array-like object args as the list of arguments.

The only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them.

So these two calls are almost equivalent:

```
func.call(context, ...args);  
func.apply(context, args);
```

They perform the same call of func with given context and arguments.
There's only a subtle difference regarding args:

The spread syntax ... allows to pass iterable args as the list to call.
The apply accepts only array-like args.

...And for objects that are both iterable and array-like, such as a real array, we can use any of them, but apply will probably be faster, because most JavaScript engines internally optimize it better.

Passing all arguments along with the context to another function is called call forwarding.

That's the simplest form of it:

```
let wrapper = function() {  
  return func.apply(this, arguments);  
};
```

When an external code calls such wrapper, it is indistinguishable from the call of the original function func.

Borrowing a method Now let's make one more minor improvement in the hashing function:

```
function hash(args) {  
  return args[0] + ',' + args[1];  
}
```

As of now, it works only on two arguments. It would be better if it could glue any number of args.

The natural solution would be to use `arr.join` method:

```
function hash(args) {  
  return args.join();  
}
```

...Unfortunately, that won't work. Because we are calling `hash(arguments)`, and `arguments` object is both iterable and array-like, but not a real array. So calling `join` on it would fail, as we can see below:

```
function hash() {  
  alert( arguments.join() ); // Error: arguments.join is not a function  
}  
  
hash(1, 2);
```

Still, there's an easy way to use array `join`:

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}

hash(1, 2);
```

The trick is called method borrowing.

We take (borrow) a join method from a regular array (`[].join`) and use `[].join.call` to run it in the context of arguments.

Why does it work?

That's because the internal algorithm of the native method `arr.join(glue)` is very simple. Taken from the specification almost "as-is":

Let glue be the first argument or, if no arguments, then a comma `","`.

Let result be an empty string.

Append `this[0]` to result.

Append glue and `this[1]`.

Append glue and `this[2]`.

...Do so until `this.length` items are glued.

Return result.

So, technically it takes `this` and joins `this[0]`, `this[1]` ...etc together. It's intentionally written in a way that allows any array-like `this` (not a coincidence, many methods follow this practice). That's why it also works with `this=arguments`.

Decorators and function properties It is generally safe to replace a function or a method with a decorated one, except for one little thing. If the original function had properties on it, like `func.calledCount` or whatever, then the decorated one will not provide them.

Because that is a wrapper. So one needs to be careful if one uses them.

E.g. in the example above if `slow` function had any properties on it, then `cachingDecorator(slow)` is a wrapper without them.

Some decorators may provide their own properties. E.g. a decorator may count how many times a function was invoked and how much time it took, and expose this

information via wrapper properties.

There exists a way to create decorators that keep access to function properties, but this requires using a special Proxy object to wrap a function. We'll discuss it later in the article Proxy and Reflect.

SummaryDecorator is a wrapper around a function that alters its behavior. The main job is still carried out by the function.

Decorators can be seen as “features” or “aspects” that can be added to a function. We can add one or add many. And all this without changing its code!

To implement cachingDecorator, we studied methods:

func.call(context, arg1, arg2...) – calls func with given context and arguments.

func.apply(context, args) – calls func passing context as this and array-like args into a list of arguments.

The generic call forwarding is usually done with apply:

```
let wrapper = function() {  
  return original.apply(this, arguments);  
};
```

We also saw an example of method borrowing when we take a method from an object and call it in the context of another object. It is quite common to take array methods and apply them to arguments. The alternative is to use rest parameters object that is a real array.

There are many decorators there in the wild. Check how well you got them by solving the tasks of this chapter.

TasksSpy decoratorimportance: 5Create a decorator spy(func) that should return a wrapper that saves all calls to function in its calls property.

Every call is saved as an array of arguments.

For instance:

```
function work(a, b) {  
  alert( a + b ); // work is an arbitrary function or method  
}
```

```
work = spy(work);
```

```
work(1, 2); // 3
```



```
work(4, 5); // 9
```

```
for (let args of work.calls) {  
  alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"  
}
```

P.S. That decorator is sometimes useful for unit-testing. Its advanced form is `sinon.spy` in `Sinon.JS` library.
Open a sandbox with tests.
The wrapper returned by `spy(f)` should store all arguments and then use `f.apply` to forward the call.

```
function spy(func) {  
  
  function wrapper(...args) {  
    // using ...args instead of arguments to store "real" array in wrapper.calls  
    wrapper.calls.push(args);  
    return func.apply(this, args);  
  }  
  
  wrapper.calls = [];  
  
  return wrapper;  
}
```

Open the solution with tests in a sandbox.
Delaying decorator importance: 5
Create a decorator `delay(f, ms)` that delays each call of `f` by `ms` milliseconds.
For instance:

```
function f(x) {  
  alert(x);  
}  
  
// create wrappers  
let f1000 = delay(f, 1000);  
let f1500 = delay(f, 1500);
```

```
f1000("test"); // shows "test" after 1000ms  
f1500("test"); // shows "test" after 1500ms
```

In other words, `delay(f, ms)` returns a "delayed by ms" variant of `f`.
In the code above, `f` is a function of a single argument, but your solution should pass all arguments and the context `this`.
Open a sandbox with tests.[solution](#)The solution:

```
function delay(f, ms) {  
  return function() {  
    setTimeout(() => f.apply(this, arguments), ms);  
  };  
}  
  
let f1000 = delay(alert, 1000);  
  
f1000("test"); // shows "test" after 1000ms
```

Please note how an arrow function is used here. As we know, arrow functions do not have own `this` and arguments, so `f.apply(this, arguments)` takes `this` and arguments from the wrapper.
If we pass a regular function, `setTimeout` would call it without arguments and `this=window` (assuming we're in the browser).
We still can pass the right `this` by using an intermediate variable, but that's a little bit more cumbersome:

```

function delay(f, ms) {

return function(...args) {
  let savedThis = this; // store this into an intermediate variable
  setTimeout(function() {
    f.apply(savedThis, args); // use it here
  }, ms);
};

}

```

Open the solution with tests in a sandbox. Debounce decorator importance: 5 The result of `debounce(f, ms)` decorator is a wrapper that suspends calls to `f` until there's `ms` milliseconds of inactivity (no calls, "cooldown period"), then invokes `f` once with the latest arguments.

In other words, debounce is like a secretary that accepts "phone calls", and waits until there's `ms` milliseconds of being quiet. And only then it transfers the latest call information to "the boss" (calls the actual `f`).

For instance, we had a function `f` and replaced it with `f = debounce(f, 1000)`.

Then if the wrapped function is called at 0ms, 200ms and 500ms, and then there are no calls, then the actual `f` will be only called once, at 1500ms. That is: after the cooldown period of 1000ms from the last call.

...And it will get the arguments of the very last call, other calls are ignored. Here's the code for it (uses the debounce decorator from the Lodash library):

```

let f = _.debounce(alert, 1000);

f("a");
setTimeout( () => f("b"), 200);
setTimeout( () => f("c"), 500);
// debounced function waits 1000ms after the last call and then runs: alert("c")

```

Now a practical example. Let's say, the user types something, and we'd like to send a request to the server when the input is finished.

There's no point in sending the request for every character typed. Instead we'd like to wait, and then process the whole result.

In a web-browser, we can setup an event handler – a function that's called on every change of an input field. Normally, an event handler is called very often, for every typed key. But if we debounce it by 1000ms, then it will be only called once, after 1000ms after the last input.

In this live example, the handler puts the result into a box below, try it:

See? The second input calls the debounced function, so its content is processed after 1000ms from the last input.

So, debounce is a great way to process a sequence of events: be it a sequence of key presses, mouse movements or something else.

It waits the given time after the last call, and then runs its function, that can process the result.

The task is to implement debounce decorator.

Hint: that's just a few lines if you think about it :)

Open a sandbox with tests.solution

```
function debounce(func, ms) {  
  let timeout;  
  return function() {  
    clearTimeout(timeout);  
    timeout = setTimeout(() => func.apply(this, arguments), ms);  
  };  
}
```

A call to debounce returns a wrapper. When called, it schedules the original function call after given ms and cancels the previous such timeout.

Open the solution with tests in a sandbox. Throttle decorator importance: 5 Create a “throttling” decorator throttle(f, ms) – that returns a wrapper.

When it's called multiple times, it passes the call to f at maximum once per ms milliseconds.

Compared to the debounce decorator, the behavior is completely different:

debounce runs the function once after the “cooldown” period. Good for processing the final result.

throttle runs it not more often than given ms time. Good for regular updates that shouldn't be very often.

In other words, throttle is like a secretary that accepts phone calls, but bothers the boss

(calls the actual f) not more often than once per ms milliseconds.

Let's check the real-life application to better understand that requirement and to see where it comes from.

For instance, we want to track mouse movements.

In a browser we can setup a function to run at every mouse movement and get the pointer location as it moves. During an active mouse usage, this function usually runs very frequently, can be something like 100 times per second (every 10 ms).

We'd like to update some information on the web-page when the pointer moves.

...But updating function update() is too heavy to do it on every micro-movement. There is also no sense in updating more often than once per 100ms.

So we'll wrap it into the decorator: use throttle(update, 100) as the function to run on each mouse move instead of the original update(). The decorator will be called often, but forward the call to update() at maximum once per 100ms.

Visually, it will look like this:

For the first mouse movement the decorated variant immediately passes the call to update. That's important, the user sees our reaction to their move immediately.

Then as the mouse moves on, until 100ms nothing happens. The decorated variant ignores calls.

At the end of 100ms – one more update happens with the last coordinates.

Then, finally, the mouse stops somewhere. The decorated variant waits until 100ms expire and then runs update with last coordinates. So, quite important, the final mouse coordinates are processed.

A code example:

```
function f(a) {  
  console.log(a);  
}
```

```
// f1000 passes calls to f at maximum once per 1000 ms
```

```
let f1000 = throttle(f, 1000);
```

```
f1000(1); // shows 1
```

```
f1000(2); // (throttling, 1000ms not out yet)
```

```
f1000(3); // (throttling, 1000ms not out yet)
```

```
// when 1000 ms time out...
```

```
// ...outputs 3, intermediate value 2 was ignored
```

P.S. Arguments and the context this passed to f1000 should be passed to the

original f.
Open a sandbox with tests.solution

```
function throttle(func, ms) {  
  
  let isThrottled = false,  
      savedArgs,  
      savedThis;  
  
  function wrapper() {  
  
    if (isThrottled) { // (2)  
      savedArgs = arguments;  
      savedThis = this;  
      return;  
    }  
    isThrottled = true;  
  
    func.apply(this, arguments); // (1)  
  
    setTimeout(function() {  
      isThrottled = false; // (3)  
      if (savedArgs) {  
        wrapper.apply(savedThis, savedArgs);  
        savedArgs = savedThis = null;  
      }  
    }, ms);  
  }  
  
  return wrapper;  
}
```

A call to throttle(func, ms) returns wrapper.

During the first call, the wrapper just runs func and sets the cooldown state (isThrottled = true).

In this state all calls are memorized in savedArgs/savedThis. Please note that both the context and the arguments are equally important and should be memorized. We need them simultaneously to reproduce the call.

After ms milliseconds pass, setTimeout triggers. The cooldown state is removed (isThrottled = false) and, if we had ignored calls, wrapper is executed with the last memorized arguments and context.

The 3rd step runs not func, but wrapper, because we not only need to execute func, but once again enter the cooldown state and setup the timeout to reset it.

Open the solution with tests in a sandbox. Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/vcall-apply-decorators","identifier":"vcall-apply-decorators"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageAdvanced working with functions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Advanced working with functions","item":"https://javascript.info/advanced-functions"}]}October 14, 2022Function bindingWhen passing object methods as callbacks, for instance to setTimeout, there's a known problem: "losing this".

In this chapter we'll see the ways to fix it.

Losing “this”We've already seen examples of losing this. Once a method is passed somewhere separately from the object – this is lost.

Here's how it may happen with setTimeout:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

As we can see, the output shows not “John” as `this.firstName`, but `undefined`! That’s because `setTimeout` got the function `user.sayHi`, separately from the object. The last line can be rewritten as:

```
let f = user.sayHi;
setTimeout(f, 1000); // lost user context
```

The method `setTimeout` in-browser is a little special: it sets `this=window` for the function call (for Node.js, this becomes the timer object, but doesn’t really matter here). So for `this.firstName` it tries to get `window.firstName`, which does not exist. In other similar cases, usually this just becomes `undefined`.

The task is quite typical – we want to pass an object method somewhere else (here – to the scheduler) where it will be called. How to make sure that it will be called in the right context?

Solution 1: a wrapper The simplest solution is to use a wrapping function:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```


Now it works, because it receives user from the outer lexical environment, and then calls the method normally.
The same, but shorter:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Looks fine, but a slight vulnerability appears in our code structure.
What if before setTimeout triggers (there's one second delay!) user changes value?
Then, suddenly, it will call the wrong object!

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...the value of user changes within 1 second
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};

// Another user in setTimeout!
```

The next solution guarantees that such thing won't happen.
Solution 2: bindFunctions provide a built-in method bind that allows to fix this.

The basic syntax is:

```
// more complex syntax will come a little later  
let boundFunc = func.bind(context);
```

The result of `func.bind(context)` is a special function-like “exotic object”, that is callable as function and transparently passes the call to `func` setting `this=context`. In other words, calling `boundFunc` is like `func` with fixed `this`. For instance, here `funcUser` passes a call to `func` with `this=user`:

```
let user = {  
  firstName: "John"  
};  
  
function func() {  
  alert(this.firstName);  
}  
  
let funcUser = func.bind(user);  
funcUser(); // John
```

Here `func.bind(user)` as a “bound variant” of `func`, with fixed `this=user`. All arguments are passed to the original `func` “as is”, for instance:

```
    let user = {
      firstName: "John"
    };

    function func(phrase) {
      alert(phrase + ', ' + this.firstName);
    }

    // bind this to user
    let funcUser = func.bind(user);

    funcUser("Hello"); // Hello, John (argument "Hello" is passed, and this=user)
```

Now let's try with an object method:

```
    let user = {
      firstName: "John",
      sayHi() {
        alert(`Hello, ${this.firstName}!`);
      }
    };

    let sayHi = user.sayHi.bind(user); // (*)

    // can run it without an object
    sayHi(); // Hello, John!
```

```
setTimeout(sayHi, 1000); // Hello, John!
```

```
// even if the value of user changes within 1 second  
// sayHi uses the pre-bound value which is reference to the old user object  
user = {  
  sayHi() { alert("Another user in setTimeout!"); }  
};
```

In the line (*) we take the method `user.sayHi` and bind it to `user`. The `sayHi` is a “bound” function, that can be called alone or passed to `setTimeout` – doesn’t matter, the context will be right.

Here we can see that arguments are passed “as is”, only this is fixed by `bind`:

```
let user = {  
  firstName: "John",  
  say(phrase) {  
    alert(`${phrase}, ${this.firstName}!`);  
  }  
};
```

```
let say = user.say.bind(user);
```

```
say("Hello"); // Hello, John! ("Hello" argument is passed to say)  
say("Bye"); // Bye, John! ("Bye" is passed to say)
```

Convenience method: `bindAll`

If an object has many methods and we plan to actively pass it around, then we could bind them all in a loop:

```

    for (let key in user) {
    if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
    }
    }

```

JavaScript libraries also provide functions for convenient mass binding , e.g. `_.bindAll(object, methodNames)` in lodash.

Partial functions Until now we have only been talking about binding this. Let's take it a step further.

We can bind not only this, but also arguments. That's rarely done, but sometimes can be handy.

The full syntax of bind:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

It allows to bind context as this and starting arguments of the function. For instance, we have a multiplication function `mul(a, b)`:

```

function mul(a, b) {
return a * b;
}

```

Let's use bind to create a function double on its base:

```
function mul(a, b) {  
  return a * b;  
}
```

```
let double = mul.bind(null, 2);
```

```
alert( double(3) ); // = mul(2, 3) = 6  
alert( double(4) ); // = mul(2, 4) = 8  
alert( double(5) ); // = mul(2, 5) = 10
```

The call to `mul.bind(null, 2)` creates a new function `double` that passes calls to `mul`, fixing `null` as the context and `2` as the first argument. Further arguments are passed “as is”.

That’s called partial function application – we create a new function by fixing some parameters of the existing one.

Please note that we actually don’t use this here. But `bind` requires it, so we must put in something like `null`.

The function `triple` in the code below triples the value:

```
function mul(a, b) {  
  return a * b;  
}
```

```
let triple = mul.bind(null, 3);
```

```
alert( triple(3) ); // = mul(3, 3) = 9  
alert( triple(4) ); // = mul(3, 4) = 12
```

```
alert( triple(5) ); // = mul(3, 5) = 15
```

Why do we usually make a partial function?

The benefit is that we can create an independent function with a readable name (double, triple). We can use it and not provide the first argument every time as it's fixed with bind.

In other cases, partial application is useful when we have a very generic function and want a less universal variant of it for convenience.

For instance, we have a function send(from, to, text). Then, inside a user object we may want to use a partial variant of it: sendTo(to, text) that sends from the current user.

Going partial without contextWhat if we'd like to fix some arguments, but not the context this? For example, for an object method.

The native bind does not allow that. We can't just omit the context and jump to arguments.

Fortunately, a function partial for binding only arguments can be easily implemented. Like this:

```
function partial(func, ...argsBound) {  
  return function(...args) { // (*)  
    return func.call(this, ...argsBound, ...args);  
  }  
}
```

```
// Usage:  
let user = {  
  firstName: "John",  
  say(time, phrase) {  
    alert(`[${time}] ${this.firstName}: ${phrase}!`);  
  }  
};
```

```
// add a partial method with fixed time  
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());
```

```
user.sayNow("Hello");  
// Something like:  
// [10:00] John: Hello!
```

The result of `partial(func[, arg1, arg2...])` call is a wrapper (*) that calls `func` with:

Same this as it gets (for `user.sayNow` call it's `user`)
Then gives it `...argsBound` – arguments from the partial call ("10:00")
Then gives it `...args` – arguments given to the wrapper ("Hello")

So easy to do it with the spread syntax, right?

Also there's a ready `_.partial` implementation from `lodash` library.

Summary `Method func.bind(context, ...args)` returns a “bound variant” of function `func` that fixes the context this and first arguments if given.

Usually we apply `bind` to fix this for an object method, so that we can pass it somewhere. For example, to `setTimeout`.

When we fix some arguments of an existing function, the resulting (less universal) function is called partially applied or partial.

Partials are convenient when we don't want to repeat the same argument over and over again. Like if we have a `send(from, to)` function, and `from` should always be the same for our task, we can get a partial and go on with it.

Tasks `Bound` function as a `method importance: 5` What will be the output?

```
function f() {  
  alert( this ); // ?  
}
```

```
let user = {  
  g: f.bind(null)  
};
```

```
user.g();
```

solution The answer: `null`.


```

    function f() {
    alert( this ); // null
    }

```

```

let user = {
  g: f.bind(null)
};

```

```

user.g();

```

The context of a bound function is hard-fixed. There's just no way to further change it.

So even while we run `user.g()`, the original function is called with `this=null`.

Second bind importance: 5 Can we change this by additional binding?

What will be the output?

```

    function f() {
    alert(this.name);
    }

```

```

f = f.bind( {name: "John"} ).bind( {name: "Ann" } );

```

```

f();

```

solutionThe answer: John.

```

    function f() {
      alert(this.name);
    }

    f = f.bind( {name: "John"} ).bind( {name: "Pete"} );

    f(); // John

```

The exotic bound function object returned by `f.bind(...)` remembers the context (and arguments if provided) only at creation time.

A function cannot be re-bound.

Function property after bind importance: 5 There's a value in the property of a function. Will it change after bind? Why, or why not?

```

    function sayHi() {
      alert( this.name );
    }
    sayHi.test = 5;

    let bound = sayHi.bind({
      name: "John"
    });

    alert( bound.test ); // what will be the output? why?

```

solutionThe answer: undefined.

The result of bind is another object. It does not have the test property.

Fix a function that loses "this"importance: 5The call to askPassword() in the code below should check the password and then call user.loginOk/loginFail depending on the answer.

But it leads to an error. Why?

Fix the highlighted line for everything to start working right (other lines are not to be changed).

```
function askPassword(ok, fail) {  
  let password = prompt("Password?", "");  
  if (password == "rockstar") ok();  
  else fail();  
}  
  
let user = {  
  name: 'John',  
  
  loginOk() {  
    alert(`${this.name} logged in`);  
  },  
  
  loginFail() {  
    alert(`${this.name} failed to log in`);  
  },  
  
};  
  
askPassword(user.loginOk, user.loginFail);
```

solutionThe error occurs because ask gets functions loginOk/loginFail without the object.

When it calls them, they naturally assume this=undefined.

Let's bind the context:

```
function askPassword(ok, fail) {  
  let password = prompt("Password?", "");  
  if (password == "rockstar") ok();  
  else fail();  
}  
  
let user = {  
  name: 'John',  
  
  loginOk() {  
    alert(`${this.name} logged in`);  
  },  
  
  loginFail() {  
    alert(`${this.name} failed to log in`);  
  },  
  
};  
  
askPassword(user.loginOk.bind(user), user.loginFail.bind(user));
```

Now it works.

An alternative solution could be:

```
//...  
askPassword(() => user.loginOk(), () => user.loginFail());
```

Usually that also works and looks good.

It's a bit less reliable though in more complex situations where user variable might change after askPassword is called, but before the visitor answers and calls () => user.loginOk().

Partial application for loginimportance: 5The task is a little more complex variant of Fix a function that loses "this".

The user object was modified. Now instead of two functions loginOk/loginFail, it has a single function user.login(true/false).

What should we pass askPassword in the code below, so that it calls user.login(true) as ok and user.login(false) as fail?

```
function askPassword(ok, fail) {
  let password = prompt("Password?", "");
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to log in') );
  }
};

askPassword(?, ?); // ?
```

Your changes should only modify the highlighted fragment.
solution

Either use a wrapper function, an arrow to be concise:

```
askPassword(() => user.login(true), () => user.login(false));
```

Now it gets user from outer variables and runs it the normal way.

Or create a partial function from `user.login` that uses `user` as the context and has the correct first argument:

```
askPassword(user.login.bind(user, true), user.login.bind(user, false));
```

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/bind", "identifier": "bind" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageAdvanced working with functions{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Advanced working with functions", "item": "https://javascript.info/advanced-functions"}]}February 1, 2021Arrow functions revisitedLet's revisit arrow functions.

Arrow functions are not just a “shorthand” for writing small stuff. They have some very specific and useful features.

JavaScript is full of situations where we need to write a small function that's executed somewhere else.

For instance:

`arr.forEach(func)` – `func` is executed by `forEach` for every array item.

`setTimeout(func)` – `func` is executed by the built-in scheduler.

...there are more.

It's in the very spirit of JavaScript to create a function and pass it somewhere.

And in such functions we usually don't want to leave the current context. That's where arrow functions come in handy.

Arrow functions have no “`this`”As we remember from the chapter Object methods, “`this`”, arrow functions do not have `this`. If `this` is accessed, it is taken from the outside.

For instance, we can use it to iterate inside an object method:

```
    let group = {
    title: "Our Group",
    students: ["John", "Pete", "Alice"],

    showList() {
      this.students.forEach(
        student => alert(this.title + ': ' + student)
      );
    }
  };

group.showList();
```

Here in `forEach`, the arrow function is used, so `this.title` in it is exactly the same as in the outer method `showList`. That is: `group.title`.
If we used a “regular” function, there would be an error:

```
    let group = {
    title: "Our Group",
    students: ["John", "Pete", "Alice"],

    showList() {
```

```

    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student);
    });
  }
};

group.showList();

```

The error occurs because `forEach` runs functions with `this=undefined` by default, so the attempt to access `undefined.title` is made. That doesn't affect arrow functions, because they just don't have `this`.

Arrow functions can't run with `new`

Not having `this` naturally means another limitation: arrow functions can't be used as constructors. They can't be called with `new`.

Arrow functions VS `bind`

There's a subtle difference between an arrow function `=>` and a regular function called with `.bind(this)`:

`.bind(this)` creates a “bound version” of the function.

The arrow `=>` doesn't create any binding. The function simply doesn't have `this`. The lookup of `this` is made exactly the same way as a regular variable search: in the outer lexical environment.

Arrows have no “arguments” Arrow functions also have no arguments variable.

That's great for decorators, when we need to forward a call with the current `this` and arguments.

For instance, `defer(f, ms)` gets a function and returns a wrapper around it that delays the call by `ms` milliseconds:


```

    function defer(f, ms) {
    return function() {
        setTimeout(() => f.apply(this, arguments), ms);
    };
}

function sayHi(who) {
    alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // Hello, John after 2 seconds

```

The same without an arrow function would look like:

```

    function defer(f, ms) {
    return function(...args) {
        let ctx = this;
        setTimeout(function() {
            return f.apply(ctx, args);
        }, ms);
    };
}

```

Here we had to create additional variables args and ctx so that the function inside setTimeout could take them.

SummaryArrow functions:

Do not have this

Do not have arguments

Can't be called with new

They also don't have super, but we didn't study it yet. We will on the chapter Class inheritance

That's because they are meant for short pieces of code that do not have their own "context", but rather work in the current one. And they really shine in that use case.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue

or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...).
`var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/arrow-functions", "identifier": "V
arrow-functions" }); };
var DisqusShortname = "javascriptinfo";
var DisqusEnabled = true;
TutorialThe JavaScript language { "@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":
[{ "@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/" },
{ "@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://
javascript.info/js" }] }
Object properties configuration
In this section we return to objects and study their properties even more in-depth.`

Property flags and descriptors
Property getters and setters
Ctrl + ! • Previous lesson
Ctrl + ! • Next lesson
Share
Tutorial map

TutorialThe JavaScript language
Object properties configuration { "@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":
[{ "@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/" },
{ "@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://
javascript.info/js" }, { "@type": "ListItem", "position": 3, "name": "Object properties
configuration", "item": "https://javascript.info/object-properties" }] }
October 18, 2022
Property flags and descriptors
As we know, objects can store properties.

Until now, a property was a simple “key-value” pair to us. But an object property is actually a more flexible and powerful thing.

In this chapter we'll study additional configuration options, and in the next we'll see how to invisibly turn them into getter/setter functions.

Property flags
Object properties, besides a value, have three special attributes (so-called “flags”):

writable – if true, the value can be changed, otherwise it's read-only.

enumerable – if true, then listed in loops, otherwise not listed.

configurable – if true, the property can be deleted and these attributes can be modified, otherwise not.

We didn't see them yet, because generally they do not show up. When we create a property “the usual way”, all of them are true. But we also can change them anytime. First, let's see how to get those flags.

The method `Object.getOwnPropertyDescriptor` allows to query the full information about a property.

The syntax is:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj

The object to get information from.

propertyName

The name of the property.

The returned value is a so-called “property descriptor” object: it contains the value and all the flags.

For instance:

```
    let user = {  
      name: "John"  
    };  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2 ) );  
/* property descriptor:  
{  
  "value": "John",  
  "writable": true,  
  "enumerable": true,  
  "configurable": true  
}  
*/
```

To change the flags, we can use `Object.defineProperty`.
The syntax is:

`Object.defineProperty(obj, propertyName, descriptor)`

`obj, propertyName`

The object and its property to apply the descriptor.

`descriptor`

Property descriptor object to apply.

If the property exists, `defineProperty` updates its flags. Otherwise, it creates the property with the given value and flags; in that case, if a flag is not supplied, it is assumed false.

For instance, here a property name is created with all falsy flags:

```
let user = {};
```

```
Object.defineProperty(user, "name", {  
  value: "John"  
});
```

```
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
```

```
alert( JSON.stringify(descriptor, null, 2 ) );
```

```
/*  
{  
  "value": "John",  
  "writable": false,  
  "enumerable": false,  
  "configurable": false  
}  
*/
```

Compare it with “normally created” user.name above: now all flags are falsy. If that’s not what we want then we’d better set them to true in descriptor.

Now let’s see effects of the flags by example.

Non-writableLet’s make user.name non-writable (can’t be reassigned) by changing writable flag:

```
let user = {  
  name: "John"  
};  
  
Object.defineProperty(user, "name", {  
  writable: false  
});  
  
user.name = "Pete"; // Error: Cannot assign to read only property 'name'
```

Now no one can change the name of our user, unless they apply their own defineProperty to override ours.

Errors appear only in strict mode

In non-strict mode, no errors occur when writing to non-writable properties and such. But the operation still won’t succeed. Flag-violating actions are just silently ignored in non-strict.

Here’s the same example, but the property is created from scratch:

```

    let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  // for new properties we need to explicitly list what's true
  enumerable: true,
  configurable: true
});

alert(user.name); // John
user.name = "Pete"; // Error

```

Non-enumerable Now let's add a custom toString to user. Normally, a built-in toString for objects is non-enumerable, it does not show up in for..in. But if we add a toString of our own, then by default it shows up in for..in, like this:

```

    let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// By default, both our properties are listed:
for (let key in user) alert(key); // name, toString

```

If we don't like it, then we can set enumerable:false. Then it won't appear in a for..in

loop, just like the built-in one:

```
    let user = {  
    name: "John",  
    toString() {  
        return this.name;  
    }  
};
```

```
Object.defineProperty(user, "toString", {  
    enumerable: false  
});
```

```
// Now our toString disappears:  
for (let key in user) alert(key); // name
```

Non-enumerable properties are also excluded from `Object.keys`:

```
alert(Object.keys(user)); // name
```

Non-configurable The non-configurable flag (`configurable:false`) is sometimes preset for built-in objects and properties.

A non-configurable property can't be deleted, its attributes can't be modified. For instance, `Math.PI` is non-writable, non-enumerable and non-configurable:

```
    let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');  
    alert( JSON.stringify(descriptor, null, 2 ) );  
    /*  
    {  
      "value": 3.141592653589793,  
      "writable": false,  
      "enumerable": false,  
      "configurable": false  
    }  
    */
```

So, a programmer is unable to change the value of Math.PI or overwrite it.

```
    Math.PI = 3; // Error, because it has writable: false  
    // delete Math.PI won't work either
```

We also can't change Math.PI to be writable again:


```
// Error, because of configurable: false
Object.defineProperty(Math, "PI", { writable: true });
```

There's absolutely nothing we can do with Math.PI.
Making a property non-configurable is a one-way road. We cannot change it back with `defineProperty`.
Please note: `configurable: false` prevents changes of property flags and its deletion, while allowing to change its value.
Here `user.name` is non-configurable, but we can still change it (as it's writable):

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  configurable: false
});

user.name = "Pete"; // works fine
delete user.name; // Error
```

And here we make `user.name` a “forever sealed” constant, just like the built-in `Math.PI`:

```

    let user = {
      name: "John"
    };

    Object.defineProperty(user, "name", {
      writable: false,
      configurable: false
    });

    // won't be able to change user.name or its flags
    // all this won't work:
    user.name = "Pete";
    delete user.name;
    Object.defineProperty(user, "name", { value: "Pete" });

```

The only attribute change possible: writable true !' false

There's a minor exception about changing flags.

We can change writable: true to false for a non-configurable property, thus preventing its value modification (to add another layer of protection). Not the other way around though.

Object.definePropertiesThere's a method Object.defineProperties(obj, descriptors) that allows to define many properties at once.

The syntax is:

```

    Object.defineProperties(obj, {
      prop1: descriptor1,
      prop2: descriptor2
    });

```

```
// ...  
});
```

For instance:

```
Object.defineProperty(user, {  
  name: { value: "John", writable: false },  
  surname: { value: "Smith", writable: false },  
  // ...  
});
```

So, we can set many properties at once.

`Object.getOwnPropertyDescriptors` To get all property descriptors at once, we can use the method `Object.getOwnPropertyDescriptors(obj)`.

Together with `Object.defineProperty` it can be used as a “flags-aware” way of cloning an object:

```
let clone = Object.defineProperty({}, Object.getOwnPropertyDescriptors(obj));
```

Normally when we clone an object, we use an assignment to copy properties, like this:

```
for (let key in user) {  
  clone[key] = user[key]  
}
```

...But that does not copy flags. So if we want a “better” clone then

`Object.defineProperty` is preferred.

Another difference is that `for..in` ignores symbolic and non-enumerable properties, but

`Object.getOwnPropertyDescriptors` returns all property descriptors including symbolic and non-enumerable ones.

Sealing an object globallyProperty descriptors work at the level of individual properties. There are also methods that limit access to the whole object:

`Object.preventExtensions(obj)`

Forbids the addition of new properties to the object.

`Object.seal(obj)`

Forbids adding/removing of properties. Sets configurable: false for all existing properties.

`Object.freeze(obj)`

Forbids adding/removing/changing of properties. Sets configurable: false, writable: false for all existing properties.

And also there are tests for them:

`Object.isExtensible(obj)`

Returns false if adding properties is forbidden, otherwise true.

`Object.isSealed(obj)`

Returns true if adding/removing properties is forbidden, and all existing properties have configurable: false.

`Object.isFrozen(obj)`

Returns true if adding/removing/changing properties is forbidden, and all current properties are configurable: false, writable: false.

These methods are rarely used in practice.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/property-descriptors", "identifier": "property-descriptors" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageObject properties configuration{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":

[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},

{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Object properties

configuration", "item": "https://javascript.info/object-properties"}]]April 13, 2022Property

getters and settersThere are two kinds of object properties.

The first kind is data properties. We already know how to work with them. All properties that we've been using until now were data properties.

The second type of property is something new. It's an accessor property. They are essentially functions that execute on getting and setting a value, but look like regular properties to an external code.

Getters and setters Accessor properties are represented by “getter” and “setter” methods. In an object literal they are denoted by get and set:

```
let obj = {  
  get propName() {  
    // getter, the code executed on getting obj.propName  
  },  
  
  set propName(value) {  
    // setter, the code executed on setting obj.propName = value  
  }  
};
```

The getter works when obj.propName is read, the setter – when it is assigned. For instance, we have a user object with name and surname:

```
let user = {  
  name: "John",  
  surname: "Smith"  
};
```

Now we want to add a fullName property, that should be "John Smith". Of course, we don't want to copy-paste existing information, so we can implement it as an accessor:

```
    let user = {  
    name: "John",  
    surname: "Smith",  
  
    get fullName() {  
        return `${this.name} ${this.surname}`;  
    }  
};  
  
alert(user.fullName); // John Smith
```

From the outside, an accessor property looks like a regular one. That's the idea of accessor properties. We don't call `user.fullName` as a function, we read it normally: the getter runs behind the scenes.
As of now, `fullName` has only a getter. If we attempt to assign `user.fullName=`, there will be an error:

```
    let user = {  
    get fullName() {  
        return `...`;  
    }  
};  
  
user.fullName = "Test"; // Error (property has only a getter)
```

Let's fix it by adding a setter for `user.fullName`:

```

    let user = {
name: "John",
surname: "Smith",

get fullName() {
    return `${this.name} ${this.surname}`;
},

set fullName(value) {
    [this.name, this.surname] = value.split(" ");
}
};

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper

```

As the result, we have a “virtual” property fullName. It is readable and writable. Accessor descriptors for accessor properties are different from those for data properties. For accessor properties, there is no value or writable, but instead there are get and set functions.

That is, an accessor descriptor may have:

- get – a function without arguments, that works when a property is read,
- set – a function with one argument, that is called when the property is set,
- enumerable – same as for data properties,
- configurable – same as for data properties.

For instance, to create an accessor fullName with defineProperty, we can pass a descriptor with get and set:

```

    let user = {
      name: "John",
      surname: "Smith"
    };

    Object.defineProperty(user, 'fullName', {
      get() {
        return `${this.name} ${this.surname}`;
      },

      set(value) {
        [this.name, this.surname] = value.split(" ");
      }
    });

    alert(user.fullName); // John Smith

    for(let key in user) alert(key); // name, surname

```

Please note that a property can be either an accessor (has get/set methods) or a data property (has a value), not both.
 If we try to supply both get and value in the same descriptor, there will be an error:

```

    // Error: Invalid property descriptor.
    Object.defineProperty({}, 'prop', {

```



```

    get() {
      return 1
    },

    value: 2
  });

```

Smarter getters/setters Getters/setters can be used as wrappers over “real” property values to gain more control over operations with them.

For instance, if we want to forbid too short names for user, we can have a setter name and keep the value in a separate property `_name`:

```

    let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Name is too short...

```

So, the name is stored in `_name` property, and the access is done via getter and setter.

Technically, external code is able to access the name directly by using `user._name`. But there is a widely known convention that properties starting with an underscore `"_"` are internal and should not be touched from outside the object.

Using for compatibility One of the great uses of accessors is that they allow to take control over a “regular” data property at any moment by replacing it with a getter and a setter and tweak its behavior.

Imagine we started implementing user objects using data properties `name` and `age`:

```
function User(name, age) {  
  this.name = name;  
  this.age = age;  
}
```

```
let john = new User("John", 25);
```

```
alert( john.age ); // 25
```

...But sooner or later, things may change. Instead of `age` we may decide to store `birthday`, because it's more precise and convenient:

```
function User(name, birthday) {  
  this.name = name;  
  this.birthday = birthday;  
}
```

```
let john = new User("John", new Date(1992, 6, 1));
```

Now what to do with the old code that still uses `age` property?

We can try to find all such places and fix them, but that takes time and can be hard to do if that code is used by many other people. And besides, `age` is a nice thing to have in user, right?

Let's keep it.

Adding a getter for `age` solves the problem:

```

    function User(name, birthday) {
    this.name = name;
    this.birthday = birthday;

    // age is calculated from the current date and birthday
    Object.defineProperty(this, "age", {
    get() {
    let todayYear = new Date().getFullYear();
    return todayYear - this.birthday.getFullYear();
    }
    });
}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday ); // birthday is available
alert( john.age );     // ...as well as the age

```

Now the old code works too and we've got a nice additional property.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};

Object.assign(this.page, {"url":"https://javascript.info/property-accessors","identifier":"property-accessors"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;

TutorialThe JavaScript language{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}]}

Prototypes, inheritancePrototypal inheritanceF.prototypeNative

prototypes
Prototype methods, objects without `__proto__` Ctrl + ! • Previous lesson Ctrl + !
Next lesson Share Tutorial map

TutorialThe JavaScript language Prototypes, inheritance
{"@context":"https://schema.org", "@type":"BreadcrumbList", "itemListElement":
[{"@type":"ListItem", "position":1, "name":"Tutorial", "item":"https://javascript.info/"},
{"@type":"ListItem", "position":2, "name":"The JavaScript language", "item":"https://javascript.info/js"}, {"@type":"ListItem", "position":3, "name":"Prototypes, inheritance", "item":"https://javascript.info/prototypes"}]} May 6, 2022 Prototypal inheritance

In programming, we often want to take something and extend it.

For instance, we have a user object with its properties and methods, and want to make admin and guest as slightly modified variants of it. We'd like to reuse what we have in user, not copy/reimplement its methods, just build a new object on top of it.

Prototypal inheritance is a language feature that helps in that.

[[Prototype]] In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification), that is either null or references another object. That object is called “a prototype”:

When we read a property from object, and it's missing, JavaScript automatically takes it from the prototype. In programming, this is called “prototypal inheritance”. And soon we'll study many examples of such inheritance, as well as cooler language features built upon it.

The property [[Prototype]] is internal and hidden, but there are many ways to set it. One of them is to use the special name `__proto__`, like this:

```
let animal = {  
  eats: true  
};  
let rabbit = {  
  jumps: true  
};
```

```
rabbit.__proto__ = animal; // sets rabbit.[[Prototype]] = animal
```

Now if we read a property from rabbit, and it's missing, JavaScript will automatically take it from animal.

For instance:

```
let animal = {  
  eats: true  
};  
let rabbit = {  
  jumps: true  
};  
  
rabbit.__proto__ = animal; // (*)  
  
// we can find both properties in rabbit now:  
alert( rabbit.eats ); // true (**)  
alert( rabbit.jumps ); // true
```

Here the line (*) sets animal to be the prototype of rabbit. Then, when alert tries to read property rabbit.eats (**), it's not in rabbit, so JavaScript follows the [[Prototype]] reference and finds it in animal (look from the bottom up):

Here we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal".

So if animal has a lot of useful properties and methods, then they become automatically available in rabbit. Such properties are called "inherited".

If we have a method in animal, it can be called on rabbit:

```
    let animal = {  
    eats: true,  
    walk() {  
      alert("Animal walk");  
    }  
  };  

```

```
let rabbit = {  
  jumps: true,  
  __proto__: animal  
};  

```

```
// walk is taken from the prototype  
rabbit.walk(); // Animal walk
```

The method is automatically taken from the prototype, like this:

The prototype chain can be longer:

```
    let animal = {  
    eats: true,  
    walk() {  
      alert("Animal walk");  
    }  
  }  

```

```
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```

Now if we read something from longEar, and it's missing, JavaScript will look for it in rabbit, and then in animal.

There are only two limitations:

The references can't go in circles. JavaScript will throw an error if we try to assign __proto__ in a circle.

The value of __proto__ can be either an object or null. Other types are ignored.

Also it may be obvious, but still: there can be only one `[[Prototype]]`. An object may not inherit from two others.

`__proto__` is a historical getter/setter for `[[Prototype]]`

It's a common mistake of novice developers not to know the difference between these two.

Please note that `__proto__` is not the same as the internal `[[Prototype]]` property. It's a getter/setter for `[[Prototype]]`. Later we'll see situations where it matters, for now let's just keep it in mind, as we build our understanding of JavaScript language.

The `__proto__` property is a bit outdated. It exists for historical reasons, modern JavaScript suggests that we should use `Object.getPrototypeOf/Object.setPrototypeOf` functions instead that get/set the prototype. We'll also cover these functions later.

By the specification, `__proto__` must only be supported by browsers. In fact though, all environments including server-side support `__proto__`, so we're quite safe using it.

As the `__proto__` notation is a bit more intuitively obvious, we use it in the examples.

Writing doesn't use prototypeThe prototype is only used for reading properties.
Write/delete operations work directly with the object.
In the example below, we assign its own walk method to rabbit:

```
    let animal = {
  eats: true,
  walk() {
    /* this method won't be used by rabbit */
  }
};

let rabbit = {
  __proto__: animal

  rabbit.walk = function() {
    alert("Rabbit! Bounce-bounce!");
  };

  rabbit.walk(); // Rabbit! Bounce-bounce!
```

From now on, rabbit.walk() call finds the method immediately in the object and executes it, without using the prototype:

Accessor properties are an exception, as assignment is handled by a setter function. So writing to such a property is actually the same as calling a function. For that reason admin.fullName works correctly in the code below:


```

    let user = {
name: "John",
surname: "Smith",

set fullName(value) {
  [this.name, this.surname] = value.split(" ");
},

get fullName() {
  return `${this.name} ${this.surname}`;
}
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// setter triggers!
admin.fullName = "Alice Cooper"; // (**)

alert(admin.fullName); // Alice Cooper, state of admin modified
alert(user.fullName); // John Smith, state of user protected

```

Here in the line (*) the property `admin.fullName` has a getter in the prototype `user`, so it is called. And in the line (**) the property has a setter in the prototype, so it is called.

The value of “this” An interesting question may arise in the example above: what’s the value of `this` inside `set fullName(value)`? Where are the properties `this.name` and `this.surname` written: into `user` or `admin`?

The answer is simple: this is not affected by prototypes at all.

No matter where the method is found: in an object or its prototype. In a method call, this is always the object before the dot.

So, the setter call `admin.fullName=` uses `admin` as `this`, not `user`.

That is actually a super-important thing, because we may have a big object with many methods, and have objects that inherit from it. And when the inheriting objects run the inherited methods, they will modify only their own states, not the state of the big object.

For instance, here `animal` represents a “method storage”, and `rabbit` makes use of it.

The call `rabbit.sleep()` sets `this.isSleeping` on the `rabbit` object:

```
// animal has methods
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// modifies rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (no such property in the prototype)
```

The resulting picture:

If we had other objects, like bird, snake, etc., inheriting from animal, they would also gain access to methods of animal. But this in each method call would be the corresponding object, evaluated at the call-time (before dot), not animal. So when we write data into this, it is stored into these objects.

As a result, methods are shared, but the object state is not.

for...in loopThe for..in loop iterates over inherited properties too.

For instance:

```
let animal = {  
  eats: true  
};
```

```
let rabbit = {  
  jumps: true,  
  __proto__: animal  
};
```

// Object.keys only returns own keys

alert(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys

for(let prop in rabbit) alert(prop); // jumps, then eats

If that's not what we want, and we'd like to exclude inherited properties, there's a built-in method `obj.hasOwnProperty(key)`: it returns true if obj has its own (not inherited) property named key.

So we can filter out inherited properties (or do something else with them):

```

    let animal = {
      eats: true
    };

    let rabbit = {
      jumps: true,
      __proto__: animal
    };

    for(let prop in rabbit) {
      let isOwn = rabbit.hasOwnProperty(prop);

      if (isOwn) {
        alert(`Our: ${prop}`); // Our: jumps
      } else {
        alert(`Inherited: ${prop}`); // Inherited: eats
      }
    }
  }
}

```

Here we have the following inheritance chain: rabbit inherits from animal, that inherits from Object.prototype (because animal is a literal object {...}, so it's by default), and then null above it:

Note, there's one funny thing. Where is the method rabbit.hasOwnProperty coming from? We did not define it. Looking at the chain we can see that the method is provided by Object.prototype.hasOwnProperty. In other words, it's inherited. ...But why does hasOwnProperty not appear in the for..in loop like eats and jumps do, if for..in lists inherited properties? The answer is simple: it's not enumerable. Just like all other properties of

Object.prototype, it has enumerable:false flag. And for..in only lists enumerable properties. That's why it and the rest of the Object.prototype properties are not listed.

Almost all other key/value-getting methods ignore inherited properties

Almost all other key/value-getting methods, such as Object.keys, Object.values and so on ignore inherited properties.

They only operate on the object itself. Properties from the prototype are not taken into account.

Summary

In JavaScript, all objects have a hidden [[Prototype]] property that's either another object or null.

We can use obj.__proto__ to access it (a historical getter/setter, there are other ways, to be covered soon).

The object referenced by [[Prototype]] is called a "prototype".

If we want to read a property of obj or call a method, and it doesn't exist, then JavaScript tries to find it in the prototype.

Write/delete operations act directly on the object, they don't use the prototype (assuming it's a data property, not a setter).

If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.

The for..in loop iterates over both its own and its inherited properties. All other key/value-getting methods only operate on the object itself.

TasksWorking with prototypeimportance: 5Here's the code that creates a pair of objects, then modifies them.

Which values are shown in the process?

```
let animal = {
  jumps: null
};
let rabbit = {
  __proto__: animal,
  jumps: true
};
```

```
alert( rabbit.jumps ); // ? (1)
```

```
delete rabbit.jumps;
```

```
alert( rabbit.jumps ); // ? (2)
```

```
delete animal.jumps;
```

```
alert( rabbit.jumps ); // ? (3)
```

There should be 3 answers.

solution

true, taken from rabbit.

null, taken from animal.

undefined, there's no such property any more.

Searching algorithm importance: 5
The task has two parts.
Given the following objects:

```
let head = {  
  glasses: 1  
};
```

```
let table = {  
  pen: 3  
};
```

```
let bed = {  
  sheet: 1,  
  pillow: 2  
};
```

```
let pockets = {  
  money: 2000  
};
```

Use `__proto__` to assign prototypes in a way that any property lookup will follow the path: `pockets !' bed !' table !' head`. For instance, `pockets.pen` should be 3 (found in table), and `bed.glasses` should be 1 (found in head).

Answer the question: is it faster to get glasses as `pockets.glasses` or `head.glasses`?
Benchmark if needed.

solution

Let's add `__proto__`:

```
    let head = {  
      glasses: 1  
    };  
  
    let table = {  
      pen: 3,  
      __proto__: head  
    };  
  
    let bed = {  
      sheet: 1,  
      pillow: 2,  
      __proto__: table  
    };  
  
    let pockets = {  
      money: 2000,  
      __proto__: bed  
    };  
  
    alert( pockets.pen ); // 3  
    alert( bed.glasses ); // 1  
    alert( table.money ); // undefined
```

In modern engines, performance-wise, there's no difference whether we take a property from an object or its prototype. They remember where the property was found and reuse it in the next request.

For instance, for `pockets.glasses` they remember where they found glasses (in `head`), and next time will search right there. They are also smart enough to update internal caches if something changes, so that optimization is safe.

Where does it write? importance: 5 We have rabbit inheriting from animal.
If we call rabbit.eat(), which object receives the full property: animal or rabbit?

```
let animal = {  
  eat() {  
    this.full = true;  
  }  
};
```

```
let rabbit = {  
  __proto__: animal  
};
```

```
rabbit.eat();
```

solution The answer: rabbit.

That's because this is an object before the dot, so rabbit.eat() modifies rabbit.

Property lookup and execution are two different things.

The method rabbit.eat is first found in the prototype, then executed with this=rabbit.

Why are both hamsters full? importance: 5 We have two hamsters: speedy and lazy inheriting from the general hamster object.

When we feed one of them, the other one is also full. Why? How can we fix it?

```
let hamster = {  
  stomach: [],  
  
  eat(food) {  
    this.stomach.push(food);
```



```

    }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// This one found the food
speedy.eat("apple");
alert( speedy.stomach ); // apple

// This one also has it, why? fix please.
alert( lazy.stomach ); // apple

```

solutionLet's look carefully at what's going on in the call `speedy.eat("apple")`.

The method `speedy.eat` is found in the prototype (`=hamster`), then executed with `this=speedy` (the object before the dot).

Then `this.stomach.push()` needs to find `stomach` property and call `push` on it. It looks for `stomach` in this (`=speedy`), but nothing found.

Then it follows the prototype chain and finds `stomach` in `hamster`.

Then it calls `push` on it, adding the food into the `stomach` of the prototype.

So all hamsters share a single `stomach`!

Both for `lazy.stomach.push(...)` and `speedy.stomach.push()`, the property `stomach` is found in the prototype (as it's not in the object itself), then the new data is pushed into it. Please note that such thing doesn't happen in case of a simple assignment `this.stomach=`:

```

    let hamster = {
stomach: [],

eat(food) {
    // assign to this.stomach instead of this.stomach.push
    this.stomach = [food];
}
};

let speedy = {
    __proto__: hamster
};

let lazy = {
    __proto__: hamster
};

// Speedy one found the food
speedy.eat("apple");
alert( speedy.stomach ); // apple

// Lazy one's stomach is empty
alert( lazy.stomach ); // <nothing>

```

Now all works fine, because `this.stomach=` does not perform a lookup of `stomach`. The value is written directly into this object. Also we can totally avoid the problem by making sure that each hamster has their own `stomach`:

```

    let hamster = {
      stomach: [],

      eat(food) {
        this.stomach.push(food);
      }
    };

    let speedy = {
      __proto__: hamster,
      stomach: []
    };

    let lazy = {
      __proto__: hamster,
      stomach: []
    };

    // Speedy one found the food
    speedy.eat("apple");
    alert( speedy.stomach ); // apple

    // Lazy one's stomach is empty
    alert( lazy.stomach ); // <nothing>

```

As a common solution, all properties that describe the state of a particular object, like stomach above, should be written into that object. That prevents such problems.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/prototype-inheritance", "identifier": "Vprototype-inheritance" }); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;

TutorialThe JavaScript languagePrototypes, inheritance{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://

javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Prototypes, inheritance", "item": "https://javascript.info/prototypes"}}}December 12, 2021F.prototypeRemember, new objects can be created with a constructor function, like new F().

If F.prototype is an object, then the new operator uses it to set [[Prototype]] for the new object.

Please note:

JavaScript had prototypal inheritance from the beginning. It was one of the core features of the language.

But in the old times, there was no direct access to it. The only thing that worked reliably was a "prototype" property of the constructor function, described in this chapter. So there are many scripts that still use it.

Please note that F.prototype here means a regular property named "prototype" on F. It sounds something similar to the term “prototype”, but here we really mean a regular property with this name.

Here's the example:

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal

alert( rabbit.eats ); // true
```

Setting `Rabbit.prototype = animal` literally states the following: "When a new Rabbit is created, assign its `[[Prototype]]` to animal".
That's the resulting picture:

On the picture, "prototype" is a horizontal arrow, meaning a regular property, and `[[Prototype]]` is vertical, meaning the inheritance of rabbit from animal.

`F.prototype` only used at new F time

`F.prototype` property is only used when new F is called, it assigns `[[Prototype]]` of the new object.

If, after the creation, `F.prototype` property changes (`F.prototype = <another object>`), then new objects created by new F will have another object as `[[Prototype]]`, but already existing objects keep the old one.

Default `F.prototype`, constructor property Every function has the "prototype" property even if we don't supply it.

The default "prototype" is an object with the only property constructor that points back to the function itself.

Like this:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```

We can check it:

```
function Rabbit() {}  
// by default:  
// Rabbit.prototype = { constructor: Rabbit }  
  
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Naturally, if we do nothing, the constructor property is available to all rabbits through `[[Prototype]]`:

```
function Rabbit() {}  
// by default:  
// Rabbit.prototype = { constructor: Rabbit }  
  
let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}  
  
alert(rabbit.constructor == Rabbit); // true (from prototype)
```

We can use constructor property to create a new object using the same constructor as the existing one.

Like here:

```
function Rabbit(name) {  
  this.name = name;  
  alert(name);  
}
```

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

```
let rabbit2 = new rabbit.constructor("Black Rabbit");
```

That's handy when we have an object, don't know which constructor was used for it (e.g. it comes from a 3rd party library), and we need to create another one of the same kind.

But probably the most important thing about "constructor" is that...

...JavaScript itself does not ensure the right "constructor" value.

Yes, it exists in the default "prototype" for functions, but that's all. What happens with it later – is totally on us.

In particular, if we replace the default prototype as a whole, then there will be no "constructor" in it.

For instance:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

So, to keep the right "constructor" we can choose to add/remove properties to the default "prototype" instead of overwriting it as a whole:

```
function Rabbit() {}

// Not overwrite Rabbit.prototype totally
// just add to it
Rabbit.prototype.jumps = true
// the default Rabbit.prototype.constructor is preserved
```

Or, alternatively, recreate the constructor property manually:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// now constructor is also correct, because we added it
```

Summary In this chapter we briefly described the way of setting a `[[Prototype]]` for objects created via a constructor function. Later we'll see more advanced programming patterns that rely on it.

Everything is quite simple, just a few notes to make things clear:

The `F.prototype` property (don't mistake it for `[[Prototype]]`) sets `[[Prototype]]` of new objects when `new F()` is called.

The value of `F.prototype` should be either an object or null: other values won't work. The "prototype" property only has such a special effect when set on a constructor function, and invoked with `new`.

On regular objects the prototype is nothing special:

```
let user = {  
  name: "John",  
  prototype: "Bla-bla" // no magic at all  
};
```

By default all functions have `F.prototype = { constructor: F }`, so we can get the constructor of an object by accessing its "constructor" property.

TasksChanging "prototype"importance: 5In the code below we create new `Rabbit`, and then try to modify its prototype.

In the start, we have this code:

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
alert( rabbit.eats ); // true
```

We added one more string (emphasized). What will alert show now?

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};
```

```
let rabbit = new Rabbit();
```

```
Rabbit.prototype = {};
```

```
alert( rabbit.eats ); // ?
```

...And if the code is like this (replaced one line)?

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};
```

```
let rabbit = new Rabbit();
```

```
Rabbit.prototype.eats = false;
```

```
alert( rabbit.eats ); // ?
```

And like this (replaced one line)?

```
function Rabbit() {}
```

```
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
delete rabbit.eats;  
  
alert( rabbit.eats ); // ?
```

The last variant:

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
delete Rabbit.prototype.eats;  
  
alert( rabbit.eats ); // ?
```

solutionAnswers:

true.

The assignment to `Rabbit.prototype` sets up `[[Prototype]]` for new objects, but it does not affect the existing ones.

false.

Objects are assigned by reference. The object from `Rabbit.prototype` is not duplicated, it's still a single object referenced both by `Rabbit.prototype` and by the `[[Prototype]]` of `rabbit`.

So when we change its content through one reference, it is visible through the other one.

true.

All delete operations are applied directly to the object. Here delete rabbit.eats tries to remove eats property from rabbit, but it doesn't have it. So the operation won't have any effect.

undefined.

The property eats is deleted from the prototype, it doesn't exist any more.

Create an object with the same constructor importance: 5 Imagine, we have an arbitrary object obj, created by a constructor function – we don't know which one, but we'd like to create a new object using it.

Can we do it like that?

```
let obj2 = new obj.constructor();
```

Give an example of a constructor function for obj which lets such code work right. And an example that makes it work wrong.

solution We can use such approach if we are sure that "constructor" property has the correct value.

For instance, if we don't touch the default "prototype", then this code works for sure:

```
function User(name) {  
  this.name = name;  
}
```

```
let user = new User('John');
let user2 = new user.constructor('Pete');

alert( user2.name ); // Pete (worked!)
```

It worked, because `User.prototype.constructor == User`.
...But if someone, so to speak, overwrites `User.prototype` and forgets to recreate `constructor` to reference `User`, then it would fail.
For instance:

```
function User(name) {
  this.name = name;
}
User.prototype = {}; // (*)

let user = new User('John');
let user2 = new user.constructor('Pete');

alert( user2.name ); // undefined
```

Why `user2.name` is undefined?
Here's how `new user.constructor('Pete')` works:

First, it looks for `constructor` in `user`. Nothing.
Then it follows the prototype chain. The prototype of `user` is `User.prototype`, and it also has no `constructor` (because we “forgot” to set it right!).
Going further up the chain, `User.prototype` is a plain object, its prototype is the built-in `Object.prototype`.
Finally, for the built-in `Object.prototype`, there's a built-in `Object.prototype.constructor == Object`. So it is used.

Finally, at the end, we have `let user2 = new Object('Pete')`.

Probably, that's not what we want. We'd like to create new User, not new Object. That's the outcome of the missing constructor.

(Just in case you're curious, the `new Object(...)` call converts its argument to an object.

That's a theoretical thing, in practice no one calls `new Object` with a value, and

generally we don't use `new Object` to make objects at all).

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before

commenting...If you have suggestions what to improve - please submit a GitHub issue

or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the `<code>` tag, for several lines –

wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/function-prototype", "identifier": "V

function-prototype" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled =

true;

TutorialThe JavaScript languagePrototypes, inheritance{"@context": "https://

schema.org", "@type": "BreadcrumbList", "itemListElement":

[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},

{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://

javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Prototypes,

inheritance", "item": "https://javascript.info/prototypes"}]]July 6, 2022Native

prototypesThe "prototype" property is widely used by the core of JavaScript itself. All

built-in constructor functions use it.

First we'll look at the details, and then how to use it for adding new capabilities to built-in objects.

Object.prototypeLet's say we output an empty object:

```
let obj = {};  
alert( obj ); // "[object Object]" ?
```

Where's the code that generates the string "[object Object]"? That's a built-in `toString` method, but where is it? The `obj` is empty!

...But the short notation `obj = {}` is the same as `obj = new Object()`, where `Object` is a built-in object constructor function, with its own prototype referencing a huge object with `toString` and other methods.
Here's what's going on:

When `new Object()` is called (or a literal object `{...}` is created), the `[[Prototype]]` of it is set to `Object.prototype` according to the rule that we discussed in the previous chapter:

So then when `obj.toString()` is called the method is taken from `Object.prototype`.
We can check it like this:

```
let obj = {};  
  
alert(obj.__proto__ === Object.prototype); // true  
  
alert(obj.toString === obj.__proto__.toString); //true  
alert(obj.toString === Object.prototype.toString); //true
```

Please note that there is no more `[[Prototype]]` in the chain above `Object.prototype`:

```
alert(Object.prototype.__proto__); // null
```

Other built-in prototypesOther built-in objects such as Array, Date, Function and others also keep methods in prototypes.

For instance, when we create an array [1, 2, 3], the default new Array() constructor is used internally. So Array.prototype becomes its prototype and provides methods. That's very memory-efficient.

By specification, all of the built-in prototypes have Object.prototype on the top. That's why some people say that “everything inherits from objects”.

Here's the overall picture (for 3 built-ins to fit):

Let's check the prototypes manually:

```
let arr = [1, 2, 3];

// it inherits from Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// then from Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// and null on the top.
```



```
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Some methods in prototypes may overlap, for instance, `Array.prototype` has its own `toString` that lists comma-delimited elements:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- the result of Array.prototype.toString
```

As we've seen before, `Object.prototype` has `toString` as well, but `Array.prototype` is closer in the chain, so the array variant is used.

In-browser tools like Chrome developer console also show inheritance (`console.dir` may need to be used for built-in objects):

Other built-in objects also work the same way. Even functions – they are objects of a built-in `Function` constructor, and their methods (`call`/`apply` and others) are taken from `Function.prototype`. Functions have their own `toString` too.

```
function f() {}  
  
alert(f.__proto__ == Function.prototype); // true  
alert(f.__proto__.__proto__ == Object.prototype); // true, inherit from objects
```

Primitives The most intricate thing happens with strings, numbers and booleans. As we remember, they are not objects. But if we try to access their properties, temporary wrapper objects are created using built-in constructors `String`, `Number` and `Boolean`. They provide the methods and disappear. These objects are created invisibly to us and most engines optimize them out, but the specification describes it exactly this way. Methods of these objects also reside in prototypes, available as `String.prototype`, `Number.prototype` and `Boolean.prototype`.

Values `null` and `undefined` have no object wrappers

Special values `null` and `undefined` stand apart. They have no object wrappers, so methods and properties are not available for them. And there are no corresponding prototypes either.

Changing native prototypes Native prototypes can be modified. For instance, if we add a method to `String.prototype`, it becomes available to all strings:

```
String.prototype.show = function() {  
  alert(this);  
};
```

```
"BOOM!".show(); // BOOM!
```

During the process of development, we may have ideas for new built-in methods we'd like to have, and we may be tempted to add them to native prototypes. But that is generally a bad idea.

Important:

Prototypes are global, so it's easy to get a conflict. If two libraries add a method `String.prototype.show`, then one of them will be overwriting the method of the other. So, generally, modifying a native prototype is considered a bad idea.

In modern programming, there is only one case where modifying native prototypes is approved. That's polyfilling.

Polyfilling is a term for making a substitute for a method that exists in the JavaScript specification, but is not yet supported by a particular JavaScript engine.

We may then implement it manually and populate the built-in prototype with it.

For instance:

```
if (!String.prototype.repeat) { // if there's no such method
// add it to the prototype

String.prototype.repeat = function(n) {
// repeat the string n times

// actually, the code should be a little bit more complex than that
// (the full algorithm is in the specification)
// but even an imperfect polyfill is often considered good enough
return new Array(n + 1).join(this);
};
}

alert( "La".repeat(3) ); // LaLaLa
```

Borrowing from prototypesIn the chapter Decorators and forwarding, call/apply we talked about method borrowing.

That's when we take a method from one object and copy it into another.
Some methods of native prototypes are often borrowed.
For instance, if we're making an array-like object, we may want to copy some Array methods to it.
E.g.

```
let obj = {  
  0: "Hello",  
  1: "world!",  
  length: 2,  
};  
  
obj.join = Array.prototype.join;  
  
alert( obj.join(',') ); // Hello,world!
```

It works because the internal algorithm of the built-in join method only cares about the correct indexes and the length property. It doesn't check if the object is indeed an array. Many built-in methods are like that.

Another possibility is to inherit by setting `obj.__proto__` to `Array.prototype`, so all Array methods are automatically available in `obj`.

But that's impossible if `obj` already inherits from another object. Remember, we only can inherit from one object at a time.

Borrowing methods is flexible, it allows to mix functionalities from different objects if needed.

Summary

All built-in objects follow the same pattern:

The methods are stored in the prototype (`Array.prototype`, `Object.prototype`, `Date.prototype`, etc.)

The object itself stores only the data (array items, object properties, the date)

Primitives also store methods in prototypes of wrapper objects: `Number.prototype`, `String.prototype` and `Boolean.prototype`. Only `undefined` and `null` do not have wrapper objects

Built-in prototypes can be modified or populated with new methods. But it's not recommended to change them. The only allowable case is probably when we add-in a new standard, but it's not yet supported by the JavaScript engine

TasksAdd method `"f.defer(ms)"` to functionsimportance: 5Add to the prototype of all functions the method `defer(ms)`, that runs the function after `ms` milliseconds.

After you do it, such code should work:

```
function f() {  
  alert("Hello!");  
}
```

```
f.defer(1000); // shows "Hello!" after 1 second
```

solution

```
Function.prototype.defer = function(ms) {  
  setTimeout(this, ms);  
};
```

```
function f() {  
  alert("Hello!");  
}
```

```
f.defer(1000); // shows "Hello!" after 1 sec
```

Add the decorating "defer()" to functions importance: 4 Add to the prototype of all functions the method defer(ms), that returns a wrapper, delaying the call by ms milliseconds.

Here's an example of how it should work:

```
function f(a, b) {  
  alert( a + b );  
}  
  
f.defer(1000)(1, 2); // shows 3 after 1 second
```

Please note that the arguments should be passed to the original function.
solution

```
Function.prototype.defer = function(ms) {  
  let f = this;  
  return function(...args) {  
    setTimeout(() => f.apply(this, args), ms);  
  }  
};  
  
// check it  
function f(a, b) {  
  alert( a + b );  
}  
  
f.defer(1000)(1, 2); // shows 3 after 1 sec
```

Please note: we use this in f.apply to make our decoration work for object methods.

So if the wrapper function is called as an object method, then this is passed to the original method f.

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

let user = {
  name: "John",
  sayHi() {
    alert(this.name);
  }
}

user.sayHi = user.sayHi.defer(1000);

user.sayHi();
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://javascript.info/native-prototypes","identifier":"native-prototypes"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;
TutorialThe JavaScript languagePrototypes, inheritance{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}],

```
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"},{"@type":"ListItem","position":3,"name":"Prototypes,  
inheritance","item":"https://javascript.info/prototypes"}}}June 29, 2022
```

Prototype methods, objects without `__proto__` In the first chapter of this section, we mentioned that there are modern methods to setup a prototype. Setting or reading the prototype with `obj.__proto__` is considered outdated and somewhat deprecated (moved to the so-called “Annex B” of the JavaScript standard, meant for browsers only).

The modern methods to get/set a prototype are:

`Object.getPrototypeOf(obj)` – returns the `[[Prototype]]` of `obj`.

`Object.setPrototypeOf(obj, proto)` – sets the `[[Prototype]]` of `obj` to `proto`.

The only usage of `__proto__`, that’s not frowned upon, is as a property when creating a new object: `{ __proto__: ... }`.

Although, there’s a special method for this too:

`Object.create(proto, [descriptors])` – creates an empty object with given `proto` as `[[Prototype]]` and optional property descriptors.

For instance:

```
let animal = {  
  eats: true  
};  
  
// create a new object with animal as a prototype  
let rabbit = Object.create(animal); // same as {__proto__: animal}  
  
alert(rabbit.eats); // true  
  
alert(Object.getPrototypeOf(rabbit) === animal); // true  
  
Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```


The `Object.create` method is a bit more powerful, as it has an optional second argument: property descriptors.

We can provide additional properties to the new object there, like this:

```
    let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

The descriptors are in the same format as described in the chapter [Property flags and descriptors](#).

We can use `Object.create` to perform an object cloning more powerful than copying properties in `for..in`:

```
let clone = Object.create(
  Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj)
);
```

This call makes a truly exact copy of `obj`, including all properties: enumerable and

non-enumerable, data properties and setters/getters – everything, and with the right `[[Prototype]]`.

Brief history There're so many ways to manage `[[Prototype]]`. How did that happen? Why?

That's for historical reasons.

The prototypal inheritance was in the language since its dawn, but the ways to manage it evolved over time.

The prototype property of a constructor function has worked since very ancient times. It's the oldest way to create objects with a given prototype.

Later, in the year 2012, `Object.create` appeared in the standard. It gave the ability to create objects with a given prototype, but did not provide the ability to get/set it. Some browsers implemented the non-standard `__proto__` accessor that allowed the user to get/set a prototype at any time, to give more flexibility to developers.

Later, in the year 2015, `Object.setPrototypeOf` and `Object.getPrototypeOf` were added to the standard, to perform the same functionality as `__proto__`. As `__proto__` was de-facto implemented everywhere, it was kind-of deprecated and made its way to the Annex B of the standard, that is: optional for non-browser environments.

Later, in the year 2022, it was officially allowed to use `__proto__` in object literals `{...}` (moved out of Annex B), but not as a getter/setter `obj.__proto__` (still in Annex B).

Why was `__proto__` replaced by the functions `getPrototypeOf/setPrototypeOf`?

Why was `__proto__` partially rehabilitated and its usage allowed in `{...}`, but not as a getter/setter?

That's an interesting question, requiring us to understand why `__proto__` is bad. And soon we'll get the answer.

Don't change `[[Prototype]]` on existing objects if speed matters

Technically, we can get/set `[[Prototype]]` at any time. But usually we only set it once at the object creation time and don't modify it anymore: `rabbit` inherits from `animal`, and that is not going to change.

And JavaScript engines are highly optimized for this. Changing a prototype "on-the-fly" with `Object.setPrototypeOf` or `obj.__proto__ =` is a very slow operation as it breaks internal optimizations for object property access operations. So avoid it unless you know what you're doing, or JavaScript speed totally doesn't matter for you.

"Very plain" objects As we know, objects can be used as associative arrays to store key/value pairs.

...But if we try to store user-provided keys in it (for instance, a user-entered dictionary), we can see an interesting glitch: all keys work fine except `"__proto__"`.

Check out the example:

```
let obj = {};  
  
let key = prompt("What's the key?", "__proto__");  
obj[key] = "some value";  
  
alert(obj[key]); // [object Object], not "some value"!
```

Here, if the user types in `__proto__`, the assignment in line 4 is ignored! That could surely be surprising for a non-developer, but pretty understandable for us. The `__proto__` property is special: it must be either an object or null. A string can not become a prototype. That's why an assignment a string to `__proto__` is ignored. But we didn't intend to implement such behavior, right? We want to store key/value pairs, and the key named `"__proto__"` was not properly saved. So that's a bug! Here the consequences are not terrible. But in other cases we may be storing objects instead of strings in `obj`, and then the prototype will indeed be changed. As a result, the execution will go wrong in totally unexpected ways. What's worse – usually developers do not think about such possibility at all. That makes such bugs hard to notice and even turn them into vulnerabilities, especially when JavaScript is used on server-side. Unexpected things also may happen when assigning to `obj.toString`, as it's a built-in object method. How can we avoid this problem? First, we can just switch to using `Map` for storage instead of plain objects, then everything's fine:

```
let map = new Map();
```

```
let key = prompt("What's the key?", "__proto__");
map.set(key, "some value");

alert(map.get(key)); // "some value" (as intended)
```

...But Object syntax is often more appealing, as it's more concise.
Fortunately, we can use objects, because language creators gave thought to that problem long ago.
As we know, `__proto__` is not a property of an object, but an accessor property of `Object.prototype`:

So, if `obj.__proto__` is read or set, the corresponding getter/setter is called from its prototype, and it gets/sets `[[Prototype]]`.
As it was said in the beginning of this tutorial section: `__proto__` is a way to access `[[Prototype]]`, it is not `[[Prototype]]` itself.
Now, if we intend to use an object as an associative array and be free of such problems, we can do it with a little trick:

```
let obj = Object.create(null);
// or: obj = { __proto__: null }

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```

`Object.create(null)` creates an empty object without a prototype ([[Prototype]] is null):

So, there is no inherited getter/setter for `__proto__`. Now it is processed as a regular data property, so the example above works right. We can call such objects “very plain” or “pure dictionary” objects, because they are even simpler than the regular plain object `{...}`. A downside is that such objects lack any built-in object methods, e.g. `toString`:

```
let obj = Object.create(null);  
  
alert(obj); // Error (no toString)
```

...But that’s usually fine for associative arrays. Note that most object-related methods are `Object.something(...)`, like `Object.keys(obj)` – they are not in the prototype, so they will keep working on such objects:

```
let chineseDictionary = Object.create(null);  
chineseDictionary.hello = "O`Y";
```

```
chineseDictionary.bye = "Q•%oÁ";  
  
alert(Object.keys(chineseDictionary)); // hello,bye
```

Summary

To create an object with the given prototype, use:

literal syntax: { __proto__: ... }, allows to specify multiple properties
or Object.create(proto, [descriptors]), allows to specify property descriptors.

The Object.create provides an easy way to shallow-copy an object with all descriptors:

```
let clone = Object.create(Object.getPrototypeOf(obj),  
Object.getOwnPropertyDescriptors(obj));
```

Modern methods to get/set the prototype are:

Object.getPrototypeOf(obj) – returns the [[Prototype]] of obj (same as __proto__ getter).
Object.setPrototypeOf(obj, proto) – sets the [[Prototype]] of obj to proto (same as __proto__ setter).

Getting/setting the prototype using the built-in __proto__ getter/setter isn't recommended, it's now in the Annex B of the specification.

We also covered prototype-less objects, created with Object.create(null) or {__proto__: null}.

These objects are used as dictionaries, to store any (possibly user-generated) keys. Normally, objects inherit built-in methods and __proto__ getter/setter from Object.prototype, making corresponding keys “occupied” and potentially causing side effects. With null prototype, objects are truly empty.

TasksAdd toString to the dictionaryimportance: 5There's an object dictionary, created

as `Object.create(null)`, to store any key/value pairs.
Add method `dictionary.toString()` into it, that should return a comma-delimited list of keys. Your `toString` should not show up in `for..in` over the object.
Here's how it should work:

```
let dictionary = Object.create(null);

// your code to add dictionary.toString method

// add some data
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // __proto__ is a regular property key here

// only apple and __proto__ are in the loop
for(let key in dictionary) {
  alert(key); // "apple", then "__proto__"
}

// your toString in action
alert(dictionary); // "apple,__proto__"
```

solution The method can take all enumerable keys using `Object.keys` and output their list.

To make `toString` non-enumerable, let's define it using a property descriptor. The syntax of `Object.create` allows us to provide an object with property descriptors as the second argument.

```
let dictionary = Object.create(null, {
  toString: { // define toString property
    value() { // the value is a function
```

```

    return Object.keys(this).join();
  }
}
});

dictionary.apple = "Apple";
dictionary.__proto__ = "test";

// apple and __proto__ is in the loop
for(let key in dictionary) {
  alert(key); // "apple", then "__proto__"
}

// comma-separated list of properties by toString
alert(dictionary); // "apple,__proto__"

```

When we create a property using a descriptor, its flags are false by default. So in the code above, `dictionary.toString` is non-enumerable. See the chapter Property flags and descriptors for review.

The difference between `calls` and `importance`: 5 Let's create a new rabbit object:

```

function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");

```

These calls do the same thing or not?

```

rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();

```


solutionThe first call has this == rabbit, the other ones have this equal to Rabbit.prototype, because it's actually the object before the dot.
So only the first call shows Rabbit, other ones show undefined:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert( this.name );
}

let rabbit = new Rabbit("Rabbit");

rabbit.sayHi();           // Rabbit
Rabbit.prototype.sayHi(); // undefined
Object.getPrototypeOf(rabbit).sayHi(); // undefined
rabbit.__proto__.sayHi(); // undefined
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://javascript.info/prototype-methods","identifier":"V
prototype-methods"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled =
true;
TutorialThe JavaScript language{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}],

```
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"}}ClassesClass basic syntaxClass inheritanceStatic properties and  
methodsPrivate and protected properties and methodsExtending built-in classesClass  
checking: "instanceof"MixinsCtrl + !•Previous lessonCtrl + !Next lessonShareTutorial map  
TutorialThe JavaScript languageClasses{"@context":"https://  
schema.org","@type":"BreadcrumbList","itemListElement":  
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},  
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://  
javascript.info/js"},{"@type":"ListItem","position":3,"name":"Classes","item":"https://  
javascript.info/classes"}]}December 16, 2021Class basic syntaxIn 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", 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" syntaxThe 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:

A new object is created.

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? 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:

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).

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. 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 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.

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.

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.

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 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 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 [...] 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

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 fieldsAs demonstrated in the chapter Function binding 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:

Pass a wrapper-function, such as `setTimeout(() => button.click(), 1000)`.

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 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 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.

Open a sandbox for the task. 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();

```

Open the solution in a [sandbox](#). Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a [GitHub issue](#) or a [pull request](#) instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a [sandbox](#) (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://\ javascript.info\class","identifier":"\class"}); };var disqus_shortcode = "javascriptinfo";var disqus_enabled = true;TutorialThe JavaScript languageClasses{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://

javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Classes", "item": "https://
javascript.info/classes"}}}May 12, 2022Class inheritanceClass inheritance is a way for
one class to extend another class.
So we can create new functionality on top of the existing.
The “extends” keywordLet’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:

...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.__proto__` to `Animal.prototype`. So, if a method is not found in `Rabbit.prototype`, JavaScript takes it from `Animal.prototype`.

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

The rabbit object (has no run).

Its prototype, that is `Rabbit.prototype` (has hide, but not run).

Its prototype, that is (due to extends) `Animal.prototype`, that finally has the run method.

As we can recall from the chapter Native prototypes, JavaScript itself uses prototypical inheritance for built-in objects. E.g. `Date.prototype.__proto__` 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 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, 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 With constructors it gets a little bit tricky.

Until now, Rabbit did not have its own constructor.

According to the specification, 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

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]]`

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. Here's the picture of what happens:

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);
```

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);
```

...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]] 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” 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:

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

To extend a class: `class Child extends Parent:`

That means `Child.prototype.__proto__` will be `Parent.prototype`, so methods are inherited.

When overriding a constructor:

We must call parent constructor as `super()` in Child constructor before using this.

When overriding another method:

We can use `super.method()` in a Child method to call Parent method.

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.

TasksError creating an instanceimportance: 5Here'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 clockimportance: 5We'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.

Open a sandbox for the task.[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);  
  }  
};
```

Open the solution in a sandbox.[Ctrl + !](#)•Previous lesson[Ctrl + !](#)Next lesson

Share Tutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/class-inheritance","identifier":"\class-inheritance"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languageClasses{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"@type":"ListItem","position":3,"name":"Classes","item":"https://javascript.info/classes"]}]May 3, 2022Static properties and methodsWe can also assign a method to the class as a whole. Such methods are called static.

In a class declaration, they are prepended by static keyword, like this:

```
class User {  
  static staticMethod() {  
    alert(this === User);  
  }  
}  
  
User.staticMethod(); // true
```

That actually does the same as assigning it as a property directly:

```
class User { }  
  
User.staticMethod = function() {  
  alert(this === User);  
};  
  
User.staticMethod(); // true
```

The value of this in User.staticMethod() call is the class constructor User itself (the “object before dot” rule).

Usually, static methods are used to implement functions that belong to the class as a whole, but not to any particular object of it.

For instance, we have Article objects and need a function to compare them.

A natural solution would be to add Article.compare static method:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

// usage
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare);

alert( articles[0].title ); // CSS
```

Here Article.compare method stands “above” articles, as a means to compare them. It’s not a method of an article, but rather of the whole class.

Another example would be a so-called “factory” method.

Let’s say, we need multiple ways to create an article:

Create by given parameters (title, date etc).
Create an empty article with today's date.
...or else somehow.

The first way can be implemented by the constructor. And for the second one we can make a static method of the class.
Such as Article.createTodays() here:

```
class Article {  
  constructor(title, date) {  
    this.title = title;  
    this.date = date;  
  }  
  
  static createTodays() {  
    // remember, this = Article  
    return new this("Today's digest", new Date());  
  }  
}
```

```
let article = Article.createTodays();
```

```
alert( article.title ); // Today's digest
```

Now every time we need to create a today's digest, we can call Article.createTodays(). Once again, that's not a method of an article, but a method of the whole class.

Static methods are also used in database-related classes to search/save/remove entries from the database, like this:

```
// assuming Article is a special class for managing articles
// static method to remove the article by id:
Article.remove({id: 12345});
```

Static methods aren't available for individual objects
Static methods are callable on classes, not on individual objects.
E.g. such code won't work:

```
// ...
article.createTodays(); /// Error: article.createTodays is not a function
```

Static properties

A recent addition

This is a recent addition to the language.
Examples work in the recent Chrome.

Static properties are also possible, they look like regular class properties, but prepended by static:

```
class Article {
  static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

That is the same as a direct assignment to Article:

```
Article.publisher = "Ilya Kantor";
```

Inheritance of static properties and methods Static properties and methods are inherited.

For instance, Animal.compare and Animal.planet in the code below are inherited and accessible as Rabbit.compare and Rabbit.planet:

```
class Animal {
  static planet = "Earth";

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

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}
```



```
// Inherit from Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Black Rabbit runs with speed 5.

alert(Rabbit.planet); // Earth
```

Now when we call `Rabbit.compare`, the inherited `Animal.compare` will be called. How does it work? Again, using prototypes. As you might have already guessed, `extends` gives `Rabbit` the `[[Prototype]]` reference to `Animal`.

So, `Rabbit extends Animal` creates two `[[Prototype]]` references:

`Rabbit` function prototypally inherits from `Animal` function.
`Rabbit.prototype` prototypally inherits from `Animal.prototype`.

As a result, inheritance works both for regular and static methods. Here, let's check that by code:

```

    class Animal {}
class Rabbit extends Animal {}

// for statics
alert(Rabbit.__proto__ === Animal); // true

// for regular methods
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true

```

Summary Static methods are used for the functionality that belongs to the class “as a whole”. It doesn’t relate to a concrete class instance. For example, a method for comparison `Article.compare(article1, article2)` or a factory method `Article.createToday()`. They are labeled by the word `static` in class declaration. Static properties are used when we’d like to store class-level data, also not bound to an instance. The syntax is:

```

    class MyClass {
static property = ...;

static method() {
    ...
}
}

```

Technically, static declaration is the same as assigning to the class itself:

```

    MyClass.property = ...
MyClass.method = ...

```

Static properties and methods are inherited.

For class B extends A the prototype of the class B itself points to A: B.[[Prototype]] = A. So if a field is not found in B, the search continues in A. TasksClass extends Object?importance: 3As we know, all objects normally inherit from Object.prototype and get access to “generic” object methods like hasOwnProperty etc. For instance:

```
class Rabbit {  
  constructor(name) {  
    this.name = name;  
  }  
}
```

```
let rabbit = new Rabbit("Rab");
```

```
// hasOwnProperty method is from Object.prototype  
alert( rabbit.hasOwnProperty('name') ); // true
```

But if we spell it out explicitly like "class Rabbit extends Object", then the result would be different from a simple "class Rabbit"?

What's the difference?

Here's an example of such code (it doesn't work – why? fix it?):

```
class Rabbit extends Object {  
  constructor(name) {  
    this.name = name;  
  }  
}
```

```
let rabbit = new Rabbit("Rab");
```

```
alert( rabbit.hasOwnProperty('name') ); // Error
```

solutionFirst, let's see why the latter code doesn't work.

The reason becomes obvious if we try to run it. An inheriting class constructor must call `super()`. Otherwise "this" won't be "defined".

So here's the fix:

```
class Rabbit extends Object {  
  constructor(name) {  
    super(); // need to call the parent constructor when inheriting  
    this.name = name;  
  }  
}
```

```
let rabbit = new Rabbit("Rab");
```

```
alert( rabbit.hasOwnProperty('name') ); // true
```

But that's not all yet.

Even after the fix, there's still an important difference between "class Rabbit extends Object" and class Rabbit.

As we know, the "extends" syntax sets up two prototypes:

Between "prototype" of the constructor functions (for methods).

Between the constructor functions themselves (for static methods).

In the case of class Rabbit extends Object it means:

```
class Rabbit extends Object {}  
  
alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) true  
alert( Rabbit.__proto__ === Object ); // (2) true
```

So Rabbit now provides access to the static methods of Object via Rabbit, like this:

```
class Rabbit extends Object {}  
  
// normally we call Object.getOwnPropertyNames  
alert ( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // a,b
```

But if we don't have extends Object, then Rabbit.__proto__ is not set to Object.
Here's the demo:

```

class Rabbit {}

alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) true
alert( Rabbit.__proto__ === Object ); // (2) false (!)
alert( Rabbit.__proto__ === Function.prototype ); // as any function by default

// error, no such function in Rabbit
alert ( Rabbit.getOwnPropertyNames({a: 1, b: 2})); // Error

```

So Rabbit doesn't provide access to static methods of Object in that case. By the way, Function.prototype also has "generic" function methods, like call, bind etc. They are ultimately available in both cases, because for the built-in Object constructor, Object.__proto__ === Function.prototype. Here's the picture:

So, to put it short, there are two differences:

```

class Rabbit
class Rabbit extends Object

```

—
needs to call super() in constructor

```

Rabbit.__proto__ === Function.prototype
Rabbit.__proto__ === Object

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/vstatic-properties-  
methods", "identifier": "vstatic-properties-methods" }); };  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;  
TutorialThe JavaScript languageClasses{"@context": "https://  
schema.org", "@type": "BreadcrumbList", "itemListElement":  
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Classes", "item": "https://  
javascript.info/classes"}]}
```

June 18, 2021 Private and protected properties and methods One of the most important principles of object oriented programming – delimiting internal interface from the external one.

That is “a must” practice in developing anything more complex than a “hello world” app. To understand this, let’s break away from development and turn our eyes into the real world.

Usually, devices that we’re using are quite complex. But delimiting the internal interface from the external one allows to use them without problems.

A real-life example For instance, a coffee machine. Simple from outside: a button, a display, a few holes... And, surely, the result – great coffee! :)

But inside... (a picture from the repair manual)

A lot of details. But we can use it without knowing anything.

Coffee machines are quite reliable, aren’t they? We can use one for years, and only if something goes wrong – bring it for repairs.

The secret of reliability and simplicity of a coffee machine – all details are well-tuned and hidden inside.

If we remove the protective cover from the coffee machine, then using it will be much more complex (where to press?), and dangerous (it can electrocute).

As we’ll see, in programming objects are like coffee machines.

But in order to hide inner details, we’ll use not a protective cover, but rather special syntax of the language and conventions.

Internal and external interface In object-oriented programming, properties and methods are split into two groups:

Internal interface – methods and properties, accessible from other methods of the class, but not from the outside.

External interface – methods and properties, accessible also from outside the class.

If we continue the analogy with the coffee machine – what’s hidden inside: a boiler tube,

heating element, and so on – is its internal interface.

An internal interface is used for the object to work, its details use each other. For instance, a boiler tube is attached to the heating element.

But from the outside a coffee machine is closed by the protective cover, so that no one can reach those. Details are hidden and inaccessible. We can use its features via the external interface.

So, all we need to use an object is to know its external interface. We may be completely unaware how it works inside, and that's great.

That was a general introduction.

In JavaScript, there are two types of object fields (properties and methods):

Public: accessible from anywhere. They comprise the external interface. Until now we were only using public properties and methods.

Private: accessible only from inside the class. These are for the internal interface.

In many other languages there also exist “protected” fields: accessible only from inside the class and those extending it (like private, but plus access from inheriting classes).

They are also useful for the internal interface. They are in a sense more widespread than private ones, because we usually want inheriting classes to gain access to them.

Protected fields are not implemented in JavaScript on the language level, but in practice they are very convenient, so they are emulated.

Now we'll make a coffee machine in JavaScript with all these types of properties. A coffee machine has a lot of details, we won't model them to stay simple (though we could).

Protecting “waterAmount” Let's make a simple coffee machine class first:

```
class CoffeeMachine {
  waterAmount = 0; // the amount of water inside

  constructor(power) {
    this.power = power;
    alert( `Created a coffee-machine, power: ${power}` );
  }
}
```



```
// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);

// add water
coffeeMachine.waterAmount = 200;
```

Right now the properties `waterAmount` and `power` are public. We can easily get/set them from the outside to any value.

Let's change `waterAmount` property to protected to have more control over it. For instance, we don't want anyone to set it below zero.

Protected properties are usually prefixed with an underscore `_`.

That is not enforced on the language level, but there's a well-known convention between programmers that such properties and methods should not be accessed from the outside.

So our property will be called `_waterAmount`:

```
class CoffeeMachine {
  _waterAmount = 0;

  set waterAmount(value) {
    if (value < 0) {
      value = 0;
    }
    this._waterAmount = value;
  }

  get waterAmount() {
    return this._waterAmount;
  }

  constructor(power) {
    this._power = power;
  }
}
```

```

    }
}

// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);

// add water
coffeeMachine.waterAmount = -10; // _waterAmount will become 0, not -10

```

Now the access is under control, so setting the water amount below zero becomes impossible.

Read-only “power” For power property, let’s make it read-only. It sometimes happens that a property must be set at creation time only, and then never modified.

That’s exactly the case for a coffee machine: power never changes.

To do so, we only need to make getter, but not the setter:

```

    class CoffeeMachine {
// ...

constructor(power) {
    this._power = power;
}

get power() {
    return this._power;
}

}

// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);

```

```
alert(` Power is: ${coffeeMachine.power}W`); // Power is: 100W
```

```
coffeeMachine.power = 25; // Error (no setter)
```

Getter/setter functions

Here we used getter/setter syntax.

But most of the time get.../set... functions are preferred, like this:

```
class CoffeeMachine {
  _waterAmount = 0;

  setWaterAmount(value) {
    if (value < 0) value = 0;
    this._waterAmount = value;
  }

  getWaterAmount() {
    return this._waterAmount;
  }
}

new CoffeeMachine().setWaterAmount(100);
```

That looks a bit longer, but functions are more flexible. They can accept multiple arguments (even if we don't need them right now).

On the other hand, get/set syntax is shorter, so ultimately there's no strict rule, it's up to you to decide.

Protected fields are inherited

If we inherit class MegaMachine extends CoffeeMachine, then nothing prevents us from accessing this._waterAmount or this._power from the methods of the new class. So protected fields are naturally inheritable. Unlike private ones that we'll see below.

Private “#waterLimit”

A recent addition

This is a recent addition to the language.

Not supported in JavaScript engines, or supported partially yet, requires polyfilling.

There's a finished JavaScript proposal, almost in the standard, that provides language-level support for private properties and methods.

Privates should start with #. They are only accessible from inside the class.

For instance, here's a private #waterLimit property and the water-checking private method #fixWaterAmount:

```
class CoffeeMachine {
  #waterLimit = 200;

  #fixWaterAmount(value) {
    if (value < 0) return 0;
    if (value > this.#waterLimit) return this.#waterLimit;
  }

  setWaterAmount(value) {
    this.#waterLimit = this.#fixWaterAmount(value);
  }
}

let coffeeMachine = new CoffeeMachine();

// can't access privates from outside of the class
coffeeMachine.#fixWaterAmount(123); // Error
coffeeMachine.#waterLimit = 1000; // Error
```

On the language level, # is a special sign that the field is private. We can't access it from outside or from inheriting classes.

Private fields do not conflict with public ones. We can have both private #waterAmount and public waterAmount fields at the same time.

For instance, let's make `waterAmount` an accessor for `#waterAmount`:

```
class CoffeeMachine {  
  #waterAmount = 0;  
  
  get waterAmount() {  
    return this.#waterAmount;  
  }  
  
  set waterAmount(value) {  
    if (value < 0) value = 0;  
    this.#waterAmount = value;  
  }  
}  
  
let machine = new CoffeeMachine();  
  
machine.waterAmount = 100;  
alert(machine.#waterAmount); // Error
```

Unlike protected ones, private fields are enforced by the language itself. That's a good thing.

But if we inherit from `CoffeeMachine`, then we'll have no direct access to `#waterAmount`. We'll need to rely on `waterAmount` getter/setter:

```
class MegaCoffeeMachine extends CoffeeMachine {  
  method() {  
    alert( this.#waterAmount ); // Error: can only access from CoffeeMachine  
  }  
}
```

```
}
```

In many scenarios such limitation is too severe. If we extend a CoffeeMachine, we may have legitimate reasons to access its internals. That's why protected fields are used more often, even though they are not supported by the language syntax.

Private fields are not available as `this[name]`

Private fields are special.

As we know, usually we can access fields using `this[name]`:

```
class User {  
  ...  
  sayHi() {  
    let fieldName = "name";  
    alert(`Hello, ${this[fieldName]}`);  
  }  
}
```

With private fields that's impossible: `this[#name]` doesn't work. That's a syntax limitation to ensure privacy.

Summary
In terms of OOP, delimiting of the internal interface from the external one is called encapsulation.

It gives the following benefits:

Protection for users, so that they don't shoot themselves in the foot

Imagine, there's a team of developers using a coffee machine. It was made by the "Best CoffeeMachine" company, and works fine, but a protective cover was removed. So the internal interface is exposed.

All developers are civilized – they use the coffee machine as intended. But one of them, John, decided that he's the smartest one, and made some tweaks in the coffee machine internals. So the coffee machine failed two days later.

That's surely not John's fault, but rather the person who removed the protective cover and let John do his manipulations.

The same in programming. If a user of a class will change things not intended to be changed from the outside – the consequences are unpredictable.

Supportable

The situation in programming is more complex than with a real-life coffee machine, because we don't just buy it once. The code constantly undergoes development and improvement.

If we strictly delimit the internal interface, then the developer of the class can freely change its internal properties and methods, even without informing the users.

If you're a developer of such class, it's great to know that private methods can be safely renamed, their parameters can be changed, and even removed, because no external code depends on them.

For users, when a new version comes out, it may be a total overhaul internally, but still simple to upgrade if the external interface is the same.

Hiding complexity

People adore using things that are simple. At least from outside. What's inside is a different thing.

Programmers are not an exception.

It's always convenient when implementation details are hidden, and a simple, well-documented external interface is available.

To hide an internal interface we use either protected or private properties:

Protected fields start with `_`. That's a well-known convention, not enforced at the language level. Programmers should only access a field starting with `_` from its class and classes inheriting from it.

Private fields start with `#`. JavaScript makes sure we can only access those from inside the class.

Right now, private fields are not well-supported among browsers, but can be polyfilled.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/private-protected-properties-methods", "identifier": "private-protected-properties-methods" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageClasses{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Classes", "item": "https://javascript.info/classes"}]}December 12, 2021Extending built-in classesBuilt-in classes like Array, Map and others are extendable also.

For instance, here PowerArray inherits from the native Array:

```
    // add one more method to it (can do more)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}
```

```
let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false
```

```
let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

Please note a very interesting thing. Built-in methods like filter, map and others – return new objects of exactly the inherited type PowerArray. Their internal implementation uses the object's constructor property for that. In the example above,

```
arr.constructor === PowerArray
```

When arr.filter() is called, it internally creates the new array of results using exactly arr.constructor, not basic Array. That's actually very cool, because we can keep using PowerArray methods further on the result. Even more, we can customize that behavior. We can add a special static getter Symbol.species to the class. If it exists, it should

return the constructor that JavaScript will use internally to create new entities in map, filter and so on.

If we'd like built-in methods like map or filter to return regular arrays, we can return Array in Symbol.species, like here:

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // built-in methods will use this as the constructor
  static get [Symbol.species]() {
    return Array;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter creates new array using arr.constructor[Symbol.species] as constructor
let filteredArr = arr.filter(item => item >= 10);

// filteredArr is not PowerArray, but Array
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function
```

As you can see, now .filter returns Array. So the extended functionality is not passed any further.

Other collections work similarly
Other collections, such as Map and Set, work alike. They also use Symbol.species.

No static inheritance in built-ins Built-in objects have their own static methods, for

instance `Object.keys`, `Array.isArray` etc.

As we already know, native classes extend each other. For instance, `Array` extends `Object`.

Normally, when one class extends another, both static and non-static methods are inherited. That was thoroughly explained in the article [Static properties and methods](#).

But built-in classes are an exception. They don't inherit statics from each other.

For example, both `Array` and `Date` inherit from `Object`, so their instances have methods from `Object.prototype`. But `Array.prototype` does not reference `Object`, so there's no, for instance, `Array.keys()` (or `Date.keys()`) static method.

Here's the picture structure for `Date` and `Object`:

As you can see, there's no link between `Date` and `Object`. They are independent, only `Date.prototype` inherits from `Object.prototype`.

That's an important difference of inheritance between built-in objects compared to what we get with `extends`.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/extend-natives", "identifier": "V  
extend-natives" }); };  
var DisqusShortname = "javascriptinfo";  
var DisqusEnabled = true;
```

TutorialThe JavaScript languageClasses{"@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Classes", "item": "https://  
javascript.info/classes"}]
```

December 12, 2021Class checking: `instanceof`The `instanceof` operator allows to check whether an object belongs to a certain class. It also takes inheritance into account.

Such a check may be necessary in many cases. For example, it can be used for building a polymorphic function, the one that treats arguments differently depending on their type.

The `instanceof` operatorThe syntax is:

```
obj instanceof Class
```

It returns true if obj belongs to the Class or a class inheriting from it.
For instance:

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
// is it an object of Rabbit class?  
alert( rabbit instanceof Rabbit ); // true
```

It also works with constructor functions:

```
// instead of class  
function Rabbit() {}  
  
alert( new Rabbit() instanceof Rabbit ); // true
```

...And with built-in classes like Array:

```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // true  
alert( arr instanceof Object ); // true
```

Please note that `arr` also belongs to the `Object` class. That's because `Array` prototypically inherits from `Object`.
Normally, `instanceof` examines the prototype chain for the check. We can also set a custom logic in the static method `Symbol.hasInstance`.
The algorithm of `obj instanceof Class` works roughly as follows:

If there's a static method `Symbol.hasInstance`, then just call it:
`Class[Symbol.hasInstance](obj)`. It should return either `true` or `false`, and we're done.
That's how we can customize the behavior of `instanceof`.
For example:

```
// setup instanceof check that assumes that  
// anything with canEat property is an animal  
class Animal {  
  static [Symbol.hasInstance](obj) {  
    if (obj.canEat) return true;  
  }  
}
```

```
let obj = { canEat: true };
```

```
alert(obj instanceof Animal); // true: Animal[Symbol.hasInstance](obj) is called
```

Most classes do not have `Symbol.hasInstance`. In that case, the standard logic is used: `obj instanceof Class` checks whether `Class.prototype` is equal to one of the prototypes in the `obj` prototype chain.

In other words, compare one after another:

```
    obj.__proto__ === Class.prototype?  
obj.__proto__.__proto__ === Class.prototype?  
obj.__proto__.__proto__.__proto__ === Class.prototype?  
...  
// if any answer is true, return true  
// otherwise, if we reached the end of the chain, return false
```

In the example above `rabbit.__proto__ === Rabbit.prototype`, so that gives the answer immediately.

In the case of an inheritance, the match will be at the second step:

```
    class Animal {}  
    class Rabbit extends Animal {}  
  
let rabbit = new Rabbit();  
alert(rabbit instanceof Animal); // true
```

```
// rabbit.__proto__ === Animal.prototype (no match)
// rabbit.__proto__.__proto__ === Animal.prototype (match!)
```

Here's the illustration of what rabbit instanceof Animal compares with Animal.prototype:

By the way, there's also a method `objA.isPrototypeOf(objB)`, that returns true if `objA` is somewhere in the chain of prototypes for `objB`. So the test of `obj instanceof Class` can be rephrased as `Class.prototype.isPrototypeOf(obj)`. It's funny, but the `Class` constructor itself does not participate in the check! Only the chain of prototypes and `Class.prototype` matters. That can lead to interesting consequences when a prototype property is changed after the object is created. Like here:

```
function Rabbit() {}
let rabbit = new Rabbit();

// changed the prototype
Rabbit.prototype = {};

// ...not a rabbit any more!
alert( rabbit instanceof Rabbit ); // false
```

Bonus: `Object.prototype.toString` for the type We already know that plain objects are

converted to string as [object Object]:

```
let obj = {};  
  
alert(obj); // [object Object]  
alert(obj.toString()); // the same
```

That's their implementation of toString. But there's a hidden feature that makes toString actually much more powerful than that. We can use it as an extended typeof and an alternative for instanceof.

Sounds strange? Indeed. Let's demystify.

By specification, the built-in toString can be extracted from the object and executed in the context of any other value. And its result depends on that value.

For a number, it will be [object Number]

For a boolean, it will be [object Boolean]

For null: [object Null]

For undefined: [object Undefined]

For arrays: [object Array]

...etc (customizable).

Let's demonstrate:

```
// copy toString method into a variable for convenience
let objectToString = Object.prototype.toString;
```

```
// what type is this?
let arr = [];
```

```
alert( objectToString.call(arr) ); // [object Array]
```

Here we used call as described in the chapter Decorators and forwarding, call/apply to execute the function objectToString in the context this=arr. Internally, the toString algorithm examines this and returns the corresponding result. More examples:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTagThe behavior of Object toString can be customized using a special object property Symbol.toStringTag. For instance:


```
let user = {  
  [Symbol.toStringTag]: "User"  
};  
  
alert( {}.toString.call(user) ); // [object User]
```

For most environment-specific objects, there is such a property. Here are some browser specific examples:

```
// toStringTag for the environment-specific object and class:  
alert( window[Symbol.toStringTag] ); // Window  
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest  
  
alert( {}.toString.call(window) ); // [object Window]  
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

As you can see, the result is exactly `Symbol.toStringTag` (if exists), wrapped into `[object ...]`.

At the end we have “typeof on steroids” that not only works for primitive data types, but also for built-in objects and even can be customized.

We can use `{}.toString.call` instead of `instanceof` for built-in objects when we want to get the type as a string rather than just to check.

SummaryLet’s summarize the type-checking methods that we know:

works for
returns

typeof
primitives
string

{}.toString
primitives, built-in objects, objects with Symbol.toStringTag
string

instanceof
objects
true/false

As we can see, {}.toString is technically a “more advanced” typeof.
And instanceof operator really shines when we are working with a class hierarchy and want to check for the class taking into account inheritance.
TasksStrange instanceof importance: 5In the code below, why does instanceof return true? We can easily see that a is not created by B().

```
function A() {}  
function B() {}  
  
A.prototype = B.prototype = {};  
  
let a = new A();
```

```
alert( a instanceof B ); // true
```

solution Yeah, looks strange indeed.

But instanceof does not care about the function, but rather about its prototype, that it matches against the prototype chain.

And here `a.__proto__ == B.prototype`, so instanceof returns true.

So, by the logic of instanceof, the prototype actually defines the type, not the constructor function.

Ctrl + ! • Previous lesson Ctrl + ! • Next lesson Share Tutorial map Comments read this before commenting... If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) `var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/instanceof", "identifier": "Vinstanceof" }); }; var disqus_shortcode = "javascriptinfo"; var disqus_enabled = true; Tutorial The JavaScript language Classes { "@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Classes", "item": "https://javascript.info/classes"}] } August 27, 2022 Mixins In JavaScript we can only inherit from a single object. There can be only one [[Prototype]] for an object. And a class may extend only one other class.`

But sometimes that feels limiting. For instance, we have a class `StreetSweeper` and a class `Bicycle`, and want to make their mix: a `StreetSweepingBicycle`.

Or we have a class `User` and a class `EventEmitter` that implements event generation, and we'd like to add the functionality of `EventEmitter` to `User`, so that our users can emit events.

There's a concept that can help here, called “mixins”.

As defined in Wikipedia, a mixin is a class containing methods that can be used by other classes without a need to inherit from it.

In other words, a mixin provides methods that implement a certain behavior, but we do not use it alone, we use it to add the behavior to other classes.

A mixin example The simplest way to implement a mixin in JavaScript is to make an object with useful methods, so that we can easily merge them into a prototype of any class.

For instance here the mixin `sayHiMixin` is used to add some “speech” for `User`:

```

    // mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};

// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, sayHiMixin);

// now User can say hi
new User("Dude").sayHi(); // Hello Dude!

```

There's no inheritance, but a simple method copying. So User may inherit from another class and also include the mixin to “mix-in” the additional methods, like this:

```

    class User extends Person {
    // ...
    }

Object.assign(User.prototype, sayHiMixin);

```

Mixins can make use of inheritance inside themselves.

For instance, here sayHiMixin inherits from sayMixin:

```
    let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (or we could use Object.setPrototypeOf to set the prototype
  here)

  sayHi() {
    // call parent method
    super.say(`Hello ${this.name}`); // (*)
  },
  sayBye() {
    super.say(`Bye ${this.name}`); // (*)
  }
};

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

// copy the methods
Object.assign(User.prototype, sayHiMixin);

// now User can say hi
new User("Dude").sayHi(); // Hello Dude!
```

Please note that the call to the parent method `super.say()` from `sayHiMixin` (at lines labelled with `(*)`) looks for the method in the prototype of that mixin, not the class. Here's the diagram (see the right part):

That's because methods `sayHi` and `sayBye` were initially created in `sayHiMixin`. So even though they got copied, their `[[HomeObject]]` internal property references `sayHiMixin`, as shown in the picture above.

As `super` looks for parent methods in `[[HomeObject]].[[Prototype]]`, that means it searches `sayHiMixin. [[Prototype]]`.

EventMixinNow let's make a mixin for real life.

An important feature of many browser objects (for instance) is that they can generate events. Events are a great way to "broadcast information" to anyone who wants it. So let's make a mixin that allows us to easily add event-related functions to any class/object.

The mixin will provide a method `.trigger(name, [...data])` to "generate an event" when something important happens to it. The name argument is a name of the event, optionally followed by additional arguments with event data.

Also the method `.on(name, handler)` that adds handler function as the listener to events with the given name. It will be called when an event with the given name triggers, and get the arguments from the `.trigger` call.

...And the method `.off(name, handler)` that removes the handler listener.

After adding the mixin, an object user will be able to generate an event "login" when the visitor logs in. And another object, say, calendar may want to listen for such events to load the calendar for the logged-in person.

Or, a menu can generate the event "select" when a menu item is selected, and other objects may assign handlers to react on that event. And so on.

Here's the code:

```
let eventMixin = {
```

```

/**
 * Subscribe to event, usage:
 * menu.on('select', function(item) { ... }
 */
on(eventName, handler) {
  if (!this._eventHandlers) this._eventHandlers = {};
  if (!this._eventHandlers[eventName]) {
    this._eventHandlers[eventName] = [];
  }
  this._eventHandlers[eventName].push(handler);
},

/**
 * Cancel the subscription, usage:
 * menu.off('select', handler)
 */
off(eventName, handler) {
  let handlers = this._eventHandlers?.[eventName];
  if (!handlers) return;
  for (let i = 0; i < handlers.length; i++) {
    if (handlers[i] === handler) {
      handlers.splice(i--, 1);
    }
  }
},

/**
 * Generate an event with the given name and data
 * this.trigger('select', data1, data2);
 */
trigger(eventName, ...args) {
  if (!this._eventHandlers?.[eventName]) {
    return; // no handlers for that event name
  }

  // call the handlers
  this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
};

```

`.on(eventName, handler)` – assigns function handler to run when the event with that name occurs. Technically, there's an `_eventHandlers` property that stores an array of handlers for each event name, and it just adds it to the list.

.off(eventName, handler) – removes the function from the handlers list.
.trigger(eventName, ...args) – generates the event: all handlers from
_eventHandlers[eventName] are called, with a list of arguments ...args.

Usage:

```
// Make a class
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Add the mixin with event-related methods
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// add a handler, to be called on selection:
menu.on("select", value => alert(`Value selected: ${value}`));

// triggers the event => the handler above runs and shows:
// Value selected: 123
menu.choose("123");
```

Now, if we'd like any code to react to a menu selection, we can listen for it with
menu.on(...).

And eventMixin mixin makes it easy to add such behavior to as many classes as we'd
like, without interfering with the inheritance chain.

SummaryMixin – is a generic object-oriented programming term: a class that contains
methods for other classes.

Some other languages allow multiple inheritance. JavaScript does not support multiple
inheritance, but mixins can be implemented by copying methods into prototype.

We can use mixins as a way to augment a class by adding multiple behaviors, like

event-handling as we have seen above.

Mixins may become a point of conflict if they accidentally overwrite existing class methods. So generally one should think well about the naming methods of a mixin, to minimize the probability of that happening.

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/mixins", "identifier": "Vmixins" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript language{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}]}

Error handlingError handling, "try...catch"Custom errors, extending ErrorCtrl + !•Previous lessonCtrl + !•Next lessonShareTutorial map

TutorialThe JavaScript languageError handling{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Error handling", "item": "https://javascript.info/error-handling"}]}

October 14, 2022Error handling, "try...catch"No matter how great we are at programming, sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user input, an erroneous server response, and for a thousand other reasons.

Usually, a script “dies” (immediately stops) in case of an error, printing it to console.

But there’s a syntax construct try...catch that allows us to “catch” errors so the script can, instead of dying, do something more reasonable.

The “try...catch” syntaxThe try...catch construct has two main blocks: try, and then catch:

```
try {  
  
  // code...  
  
} catch (err) {  
  
  // error handling  
  
}
```

It works like this:

First, the code in `try {...}` is executed.

If there were no errors, then `catch (err)` is ignored: the execution reaches the end of `try` and goes on, skipping `catch`.

If an error occurs, then the `try` execution is stopped, and control flows to the beginning of `catch (err)`. The `err` variable (we can use any name for it) will contain an error object with details about what happened.

So, an error inside the `try {...}` block does not kill the script – we have a chance to handle it in `catch`.

Let's look at some examples.

An errorless example: shows `alert (1)` and `(2)`:

```
try {  
    alert('Start of try runs'); // (1) <--  
    // ...no errors here  
    alert('End of try runs'); // (2) <--  
} catch (err) {  
    alert('Catch is ignored, because there are no errors'); // (3)
```

```
}
```

An example with an error: shows (1) and (3):

```
    try {  
        alert('Start of try runs'); // (1) <--  
        lalala; // error, variable is not defined!  
        alert('End of try (never reached)'); // (2)  
    } catch (err) {  
        alert(`Error has occurred!`); // (3) <--  
    }
```

try...catch only works for runtime errors

For try...catch to work, the code must be runnable. In other words, it should be valid JavaScript.

It won't work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
  {}
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called “parse-time” errors and are unrecoverable (from inside that code). That’s because the engine can’t understand the code. So, try...catch can only handle errors that occur in valid code. Such errors are called “runtime errors” or, sometimes, “exceptions”.

try...catch works synchronously
If an exception happens in “scheduled” code, like in setTimeout, then try...catch won't catch it:

```
try {
  setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

That's because the function itself is executed later, when the engine has already left the try...catch construct.

To catch an exception inside a scheduled function, try...catch must be inside that function:

```
        setTimeout(function() {  
    try {  
        noSuchVariable; // try...catch handles the error!  
    } catch {  
        alert( "error is caught here!" );  
    }  
}, 1000);
```

Error object When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to catch:

```
        try {  
    // ...  
} catch (err) { // <-- the "error object", could use another word instead of err  
    // ...  
}
```

For all built-in errors, the error object has two main properties:

name

Error name. For instance, for an undefined variable that's "ReferenceError".

message

Textual message about error details.

There are other non-standard properties available in most environments. One of most widely used and supported is:

stack

Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

For instance:

```
    try {  
      lalala; // error, variable is not defined!  
    } catch (err) {  
      alert(err.name); // ReferenceError  
      alert(err.message); // lalala is not defined  
      alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)  
  
      // Can also show an error as a whole  
      // The error is converted to string as "name: message"  
      alert(err); // ReferenceError: lalala is not defined  
    }  
}
```

Optional “catch” binding

A recent addition

This is a recent addition to the language.
Old browsers may need polyfills.

If we don't need error details, catch may omit it:

```
    try {  
    // ...  
} catch { // <-- without (err)  
    // ...  
}
```

Using “try...catch” Let’s explore a real-life use case of try...catch.
As we already know, JavaScript supports the `JSON.parse(str)` method to read JSON-encoded values.
Usually it’s used to decode data received over the network, from the server or another source.
We receive it and call `JSON.parse` like this:

```
let json = '{"name":"John", "age": 30}'; // data from the server  
  
let user = JSON.parse(json); // convert the text representation to JS object  
  
// now user is an object with properties from the string  
alert( user.name ); // John  
alert( user.age ); // 30
```

You can find more detailed information about JSON in the JSON methods, toJSON chapter.
If json is malformed, `JSON.parse` generates an error, so the script “dies”.
Should we be satisfied with that? Of course not!
This way, if something’s wrong with the data, the visitor will never know that (unless they open the developer console). And people really don’t like when something “just

dies" without any error message.
Let's use try...catch to handle the error:

```
    let json = "{ bad json }";

try {

    let user = JSON.parse(json); // <-- when an error occurs...
    alert( user.name ); // doesn't work

} catch (err) {
    // ...the execution jumps here
    alert( "Our apologies, the data has errors, we'll try to request it one more time." );
    alert( err.name );
    alert( err.message );
}
```

Here we use the catch block only to show the message, but we can do much more: send a new network request, suggest an alternative to the visitor, send information about the error to a logging facility, All much better than just dying.

Throwing our own errorsWhat if json is syntactically correct, but doesn't have a required name property?

Like this:


```

    let json = '{ "age": 30 }'; // incomplete data

try {

    let user = JSON.parse(json); // <-- no errors
    alert( user.name ); // no name!

} catch (err) {
    alert( "doesn't execute" );
}

```

Here `JSON.parse` runs normally, but the absence of name is actually an error for us. To unify error handling, we'll use the throw operator. "Throw" operatorThe throw operator generates an error. The syntax is:

```
throw <error object>
```

Technically, we can use anything as an error object. That may be even a primitive, like a number or a string, but it's better to use objects, preferably with name and message properties (to stay somewhat compatible with built-in errors). JavaScript has many built-in constructors for standard errors: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` and others. We can use them to create error objects as well. Their syntax is:

```

    let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...

```

For built-in errors (not for any objects, just for errors), the name property is exactly

the name of the constructor. And message is taken from the argument.
For instance:

```
let error = new Error("Things happen o_O");  
  
alert(error.name); // Error  
alert(error.message); // Things happen o_O
```

Let's see what kind of error JSON.parse generates:

```
try {  
  JSON.parse("{ bad json o_O }");  
} catch (err) {  
  alert(err.name); // SyntaxError  
  alert(err.message); // Unexpected token b in JSON at position 2  
}
```

As we can see, that's a `SyntaxError`.
And in our case, the absence of name is an error, as users must have a name.
So let's throw it:

```

    let json = '{ "age": 30 }'; // incomplete data

try {

    let user = JSON.parse(json); // <-- no errors

    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name"); // (*)
    }

    alert( user.name );

} catch (err) {
    alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}

```

In the line (*), the throw operator generates a `SyntaxError` with the given message, the same way as JavaScript would generate it itself. The execution of try immediately stops and the control flow jumps into catch.

Now catch became a single place for all error handling: both for `JSON.parse` and other cases.

Rethrowing In the example above we use try...catch to handle incorrect data. But is it possible that another unexpected error occurs within the try {...} block? Like a programming error (variable is not defined) or something else, not just this “incorrect data” thing.

For example:

```

    let json = '{ "age": 30 }'; // incomplete data

try {
    user = JSON.parse(json); // <-- forgot to put "let" before user

    // ...
} catch (err) {
    alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined
    // (no JSON Error actually)
}

```

Of course, everything's possible! Programmers do make mistakes. Even in open-source utilities used by millions for decades – suddenly a bug may be discovered that leads to terrible hacks.

In our case, try...catch is placed to catch “incorrect data” errors. But by its nature, catch gets all errors from try. Here it gets an unexpected error, but still shows the same “JSON Error” message. That’s wrong and also makes the code more difficult to debug. To avoid such problems, we can employ the “rethrowing” technique. The rule is simple: Catch should only process errors that it knows and “rethrow” all others. The “rethrowing” technique can be explained in more detail as:

Catch gets all errors.

In the catch (err) {...} block we analyze the error object err.

If we don’t know how to handle it, we do throw err.

Usually, we can check the error type using the instanceof operator:

```

    try {
      user = { /*...*/ };
    } catch (err) {
      if (err instanceof ReferenceError) {
        alert('ReferenceError'); // "ReferenceError" for accessing an undefined variable
      }
    }
  }
}

```

We can also get the error class name from `err.name` property. All native errors have it. Another option is to read `err.constructor.name`. In the code below, we use rethrowing so that catch only handles `SyntaxError`:

```

    let json = '{ "age": 30 }'; // incomplete data
  try {

    let user = JSON.parse(json);

    if (!user.name) {
      throw new SyntaxError("Incomplete data: no name");
    }

    blabla(); // unexpected error

    alert( user.name );

  } catch (err) {

    if (err instanceof SyntaxError) {
      alert( "JSON Error: " + err.message );
    } else {
      throw err; // rethrow (*)
    }
  }
}

```

```
}
```

The error throwing on line (*) from inside catch block “falls out” of try...catch and can be either caught by an outer try...catch construct (if it exists), or it kills the script. So the catch block actually handles only errors that it knows how to deal with and “skips” all others.

The example below demonstrates how such errors can be caught by one more level of try...catch:

```
function readData() {  
  let json = '{ "age": 30 }';  
  
  try {  
    // ...  
    blabla(); // error!  
  } catch (err) {  
    // ...  
    if (!(err instanceof SyntaxError)) {  
      throw err; // rethrow (don't know how to deal with it)  
    }  
  }  
}  
  
try {  
  readData();  
} catch (err) {  
  alert( "External catch got: " + err ); // caught it!  
}
```

Here readData only knows how to handle SyntaxError, while the outer try...catch knows how to handle everything.

try...catch...finallyWait, that's not all.

The try...catch construct may have one more code clause: finally.

If it exists, it runs in all cases:

after try, if there were no errors,

after catch, if there were errors.

The extended syntax looks like this:

```
    try {  
    ... try to execute the code ...  
} catch (err) {  
    ... handle errors ...  
} finally {  
    ... execute always ...  
}
```

Try running this code:

```
    try {  
    alert( 'try' );  
    if (confirm('Make an error?')) BAD_CODE();  
} catch (err) {  
    alert( 'catch' );  
} finally {  
    alert( 'finally' );  
}
```

The code has two ways of execution:

If you answer “Yes” to “Make an error?”, then try -> catch -> finally.
If you say “No”, then try -> finally.

The finally clause is often used when we start doing something and want to finalize it in any case of outcome.

For instance, we want to measure the time that a Fibonacci numbers function fib(n) takes. Naturally, we can start measuring before it runs and finish afterwards. But what if there's an error during the function call? In particular, the implementation of fib(n) in the code below returns an error for negative or non-integer numbers.

The finally clause is a great place to finish the measurements no matter what.

Here finally guarantees that the time will be measured correctly in both situations – in case of a successful execution of fib and in case of an error in it:

```
let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) !== n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}
```



```
alert(result || "error occurred");

alert( `execution took ${diff}ms` );
```

You can check by running the code with entering 35 into prompt – it executes normally, finally after try. And then enter -1 – there will be an immediate error, and the execution will take 0ms. Both measurements are done correctly. In other words, the function may finish with return or throw, that doesn't matter. The finally clause executes in both cases.

Variables are local inside try...catch...finally

Please note that result and diff variables in the code above are declared before try...catch.

Otherwise, if we declared let in try block, it would only be visible inside of it.

finally and return

The finally clause works for any exit from try...catch. That includes an explicit return.

In the example below, there's a return in try. In this case, finally is executed just before the control returns to the outer code.

```
function func() {

  try {
    return 1;

  } catch (err) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}
```

```
}
```

```
alert( func() ); // first works alert from finally, and then this one
```

try...finally

The try...finally construct, without catch clause, is also useful. We apply it when we don't want to handle errors here (let them fall through), but want to be sure that processes that we started are finalized.

```
function func() {  
  // start doing something that needs completion (like measurements)  
  try {  
    // ...  
  } finally {  
    // complete that thing even if all dies  
  }  
}
```

In the code above, an error inside try always falls out, because there's no catch. But finally works before the execution flow leaves the function.

Global catch

Environment-specific

The information from this section is not a part of the core JavaScript.

Let's imagine we've got a fatal error outside of try...catch, and the script died. Like a programming error or some other terrible thing.

Is there a way to react on such occurrences? We may want to log the error, show something to the user (normally they don't see error messages), etc.

There is none in the specification, but environments usually provide it, because it's really useful. For instance, Node.js has `process.on("uncaughtException")` for that. And in the browser we can assign a function to the special `window.onerror` property, that will run in case of an uncaught error.

The syntax:

```
    window.onerror = function(message, url, line, col, error) {  
    // ...  
};
```

message
Error message.
url
URL of the script where error happened.
line, col
Line and column numbers where error happened.
error
Error object.

For instance:

```
<script>  
window.onerror = function(message, url, line, col, error) {  
    alert( `${message}\n At ${line}:${col} of ${url}` );  
};  
  
function readData() {  
    badFunc(); // Whoops, something went wrong!  
}  
  
readData();  
</script>
```

The role of the global handler `window.onerror` is usually not to recover the script execution – that's probably impossible in case of programming errors, but to send the

error message to developers.

There are also web-services that provide error-logging for such cases, like <https://errorception.com> or <https://www.muscula.com>.

They work like this:

We register at the service and get a piece of JS (or a script URL) from them to insert on pages.

That JS script sets a custom `window.onerror` function.

When an error occurs, it sends a network request about it to the service.

We can log in to the service web interface and see errors.

Summary The `try...catch` construct allows to handle runtime errors. It literally allows to “try” running the code and “catch” errors that may occur in it.

The syntax is:

```
    try {  
    // run this code  
} catch (err) {  
    // if an error happened, then jump here  
    // err is the error object  
} finally {  
    // do in any case after try/catch  
}
```

There may be no catch section or no finally, so shorter constructs `try...catch` and `try...finally` are also valid.

Error objects have following properties:

`message` – the human-readable error message.

`name` – the string with error name (error constructor name).

`stack` (non-standard, but well-supported) – the stack at the moment of error creation.

If an error object is not needed, we can omit it by using `catch {` instead of `catch (err) {`.

We can also generate our own errors using the `throw` operator. Technically, the argument of `throw` can be anything, but usually it's an error object inheriting from the built-in `Error` class. More on extending errors in the next chapter.

Rethrowing is a very important pattern of error handling: a catch block usually expects and knows how to handle the particular error type, so it should rethrow errors it doesn't know.

Even if we don't have `try...catch`, most environments allow us to setup a “global” error handler to catch errors that “fall out”. In-browser, that's `window.onerror`.

TasksFinally or just the code?importance: 5Compare the two code fragments.

The first one uses finally to execute the code after try...catch:

```
    try {  
    work work  
    } catch (err) {  
    handle errors  
    } finally {  
    cleanup the working space  
    }
```

The second fragment puts the cleaning right after try...catch:

```
    try {  
    work work  
    } catch (err) {  
    handle errors  
    }
```

cleanup the working space

We definitely need the cleanup after the work, doesn't matter if there was an error or not.

Is there an advantage here in using finally or both code fragments are equal? If there is such an advantage, then give an example when it matters.

solutionThe difference becomes obvious when we look at the code inside a function.

The behavior is different if there's a "jump out" of try...catch.

For instance, when there's a return inside try...catch. The finally clause works in case of any exit from try...catch, even via the return statement: right after try...catch is done, but before the calling code gets the control.

```
function f() {  
  try {  
    alert('start');  
    return "result";  
  } catch (err) {  
    /// ...  
  } finally {  
    alert('cleanup!');  
  }  
}
```

f(); // cleanup!

...Or when there's a throw, like here:

```
function f() {  
  try {  
    alert('start');  
    throw new Error("an error");  
  } catch (err) {  
    // ...  
  }  
}
```

```

    if("can't handle the error") {
        throw err;
    }

    } finally {
        alert('cleanup!')
    }
}

f(); // cleanup!

```

It's finally that guarantees the cleanup here. If we just put the code at the end of `f`, it wouldn't run in these situations.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/try-catch","identifier":"\try-catch"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languageError handling{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}],{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},{"@type":"ListItem","position":3,"name":"Error handling","item":"https://javascript.info/error-handling"}]}June 26, 2021Custom errors, extending ErrorWhen we develop something, we often need our own error classes to reflect specific things that may go wrong in our tasks. For errors in network operations we may need `HttpError`, for database operations `DbError`, for searching operations `NotFoundError` and so on. Our errors should support basic error properties like `message`, `name` and, preferably, `stack`. But they also may have other properties of their own, e.g. `HttpError` objects may have a `statusCode` property with a value like 404 or 403 or 500. JavaScript allows to use `throw` with any argument, so technically our custom error classes don't need to inherit from `Error`. But if we inherit, then it becomes possible to use `obj instanceof Error` to identify error objects. So it's better to inherit from it. As the application grows, our own errors naturally form a hierarchy. For instance, `HttpTimeoutError` may inherit from `HttpError`, and so on. Extending ErrorAs an example, let's consider a function `readUser(json)` that should read JSON with user data. Here's an example of how a valid json may look:

```
let json = `{ "name": "John", "age": 30 }`;
```

Internally, we'll use `JSON.parse`. If it receives malformed json, then it throws `SyntaxError`. But even if json is syntactically correct, that doesn't mean that it's a valid user, right? It may miss the necessary data. For instance, it may not have name and age properties that are essential for our users.

Our function `readUser(json)` will not only read JSON, but check ("validate") the data. If there are no required fields, or the format is wrong, then that's an error. And that's not a `SyntaxError`, because the data is syntactically correct, but another kind of error. We'll call it `ValidationError` and create a class for it. An error of that kind should also carry the information about the offending field.

Our `ValidationError` class should inherit from the `Error` class.

The `Error` class is built-in, but here's its approximate code so we can understand what we're extending:

```
// The "pseudocode" for the built-in Error class defined by JavaScript itself
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (different names for different built-in error classes)
    this.stack = <call stack>; // non-standard, but most environments support it
  }
}
```

Now let's inherit `ValidationError` from it and try it in action:

```
class ValidationError extends Error {
```



```

    constructor(message) {
        super(message); // (1)
        this.name = "ValidationError"; // (2)
    }
}

function test() {
    throw new ValidationError("Whoops!");
}

try {
    test();
} catch(err) {
    alert(err.message); // Whoops!
    alert(err.name); // ValidationError
    alert(err.stack); // a list of nested calls with line numbers for each
}

```

Please note: in the line (1) we call the parent constructor. JavaScript requires us to call `super` in the child constructor, so that's obligatory. The parent constructor sets the `message` property.

The parent constructor also sets the `name` property to `"Error"`, so in the line (2) we reset it to the right value.

Let's try to use it in `readUser(json)`:

```

    class ValidationError extends Error {
    constructor(message) {
        super(message);
        this.name = "ValidationError";
    }
}

// Usage

```

```
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }

  return user;
}
```

// Working example with try..catch

```
try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No field: name
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it (**)
  }
}
```

The try..catch block in the code above handles both our ValidationError and the built-in SyntaxError from JSON.parse.

Please take a look at how we use instanceof to check for the specific error type in the line (*).

We could also look at err.name, like this:

```
// ...
// instead of (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

The instanceof version is much better, because in the future we are going to extend

ValidationError, make subtypes of it, like PropertyRequiredError. And instanceof check will continue to work for new inheriting classes. So that's future-proof. Also it's important that if catch meets an unknown error, then it rethrows it in the line (**). The catch block only knows how to handle validation and syntax errors, other kinds (caused by a typo in the code or other unknown reasons) should fall through. Further inheritance The ValidationError class is very generic. Many things may go wrong. The property may be absent or it may be in a wrong format (like a string value for age instead of a number). Let's make a more concrete class PropertyRequiredError, exactly for absent properties. It will carry additional information about the property that's missing.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}
```

```

    }

    return user;
}

// Working example with try..catch

try {
    let user = readUser('{ "age": 25 }');
} catch (err) {
    if (err instanceof ValidationError) {
        alert("Invalid data: " + err.message); // Invalid data: No property: name
        alert(err.name); // PropertyRequiredError
        alert(err.property); // name
    } else if (err instanceof SyntaxError) {
        alert("JSON Syntax Error: " + err.message);
    } else {
        throw err; // unknown error, rethrow it
    }
}

```

The new class `PropertyRequiredError` is easy to use: we only need to pass the property name: `new PropertyRequiredError(property)`. The human-readable message is generated by the constructor.

Please note that `this.name` in `PropertyRequiredError` constructor is again assigned manually. That may become a bit tedious – to assign `this.name = <class name>` in every custom error class. We can avoid it by making our own “basic error” class that assigns `this.name = this.constructor.name`. And then inherit all our custom errors from it. Let’s call it `MyError`.

Here’s the code with `MyError` and other custom error classes, simplified:

```

class MyError extends Error {
    constructor(message) {

```

```

    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.property = property;
  }
}

// name is correct
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError

```

Now custom errors are much shorter, especially `ValidationError`, as we got rid of the `"this.name = ..."` line in the constructor.

Wrapping exceptions The purpose of the function `readUser` in the code above is “to read the user data”. There may occur different kinds of errors in the process. Right now we have `SyntaxError` and `ValidationError`, but in the future `readUser` function may grow and probably generate other kinds of errors.

The code which calls `readUser` should handle these errors. Right now it uses multiple ifs in the catch block, that check the class and handle known errors and rethrow the unknown ones.

The scheme is like this:

```

    try {
    ...
    readUser() // the potential error source
    ...
  } catch (err) {
    if (err instanceof ValidationError) {
      // handle validation errors
    } else if (err instanceof SyntaxError) {
      // handle syntax errors
    } else {
      throw err; // unknown error, rethrow it
    }
  }
}

```

In the code above we can see two types of errors, but there can be more. If the `readUser` function generates several kinds of errors, then we should ask ourselves: do we really want to check for all error types one-by-one every time? Often the answer is “No”: we’d like to be “one level above all that”. We just want to know if there was a “data reading error” – why exactly it happened is often irrelevant (the error message describes it). Or, even better, we’d like to have a way to get the error details, but only if we need to.

The technique that we describe here is called “wrapping exceptions”.

We’ll make a new class `ReadError` to represent a generic “data reading” error. The function `readUser` will catch data reading errors that occur inside it, such as `ValidationError` and `SyntaxError`, and generate a `ReadError` instead. The `ReadError` object will keep the reference to the original error in its `cause` property.

Then the code that calls `readUser` will only have to check for `ReadError`, not for every kind of data reading errors. And if it needs more details of an error, it can check its `cause` property.

Here’s the code that defines `ReadError` and demonstrates its use in `readUser` and `try..catch`:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /* ... */ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
```

```

    if (!user.age) {
        throw new PropertyRequiredError("age");
    }

    if (!user.name) {
        throw new PropertyRequiredError("name");
    }
}

function readUser(json) {
    let user;

    try {
        user = JSON.parse(json);
    } catch (err) {
        if (err instanceof SyntaxError) {
            throw new ReadError("Syntax Error", err);
        } else {
            throw err;
        }
    }

    try {
        validateUser(user);
    } catch (err) {
        if (err instanceof ValidationError) {
            throw new ReadError("Validation Error", err);
        } else {
            throw err;
        }
    }
}

try {
    readUser('{bad json}');
} catch (e) {
    if (e instanceof ReadError) {
        alert(e);
        // Original error: SyntaxError: Unexpected token b in JSON at position 1
        alert("Original error: " + e.cause);
    } else {
        throw e;
    }
}

```

In the code above, readUser works exactly as described – catches syntax and validation errors and throws ReadError errors instead (unknown errors are rethrown as usual).

So the outer code checks instanceof ReadError and that's it. No need to list all possible error types.

The approach is called “wrapping exceptions”, because we take “low level” exceptions and “wrap” them into ReadError that is more abstract. It is widely used in object-oriented programming.

Summary

We can inherit from Error and other built-in error classes normally. We just need to take care of the name property and don't forget to call super.

We can use instanceof to check for particular errors. It also works with inheritance. But sometimes we have an error object coming from a 3rd-party library and there's no easy way to get its class. Then name property can be used for such checks.

Wrapping exceptions is a widespread technique: a function handles low-level exceptions and creates higher-level errors instead of various low-level ones. Low-level exceptions sometimes become properties of that object like err.cause in the examples above, but that's not strictly required.

TasksInherit from SyntaxErrorimportance: 5Create a class FormatError that inherits from the built-in SyntaxError class.

It should support message, name and stack properties.

Usage example:

```
let err = new FormatError("formatting error");

alert( err.message ); // formatting error
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof FormatError ); // true
alert( err instanceof SyntaxError ); // true (because inherits from SyntaxError)
```

solution


```

    class FormatError extends SyntaxError {
    constructor(message) {
        super(message);
        this.name = this.constructor.name;
    }
}

```

```
let err = new FormatError("formatting error");
```

```

alert( err.message ); // formatting error
alert( err.name ); // FormatError
alert( err.stack ); // stack

```

```
alert( err instanceof SyntaxError ); // true
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/custom-errors", "identifier": "V custom-errors" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript language{"@context": "https:// schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https:// javascript.info/js"}]}

Promises, async/awaitIntroduction: callbacksPromisePromises chainingError handling with promisesPromise APIPromisificationMicrotasksAsync/ awaitCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map

TutorialThe JavaScript languagePromises, async/await{"@context": "https:// schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https:// javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Promises, async/ await", "item": "https://javascript.info/async"}]}

June 18, 2022Introduction: callbacks

We use browser methods in examples here

To demonstrate the use of callbacks, promises and other abstract concepts,

we'll be using some browser methods: specifically, loading scripts and performing simple document manipulations.

If you're not familiar with these methods, and their usage in the examples is confusing, you may want to read a few chapters from the next part of the tutorial.

Although, we'll try to make things clear anyway. There won't be anything really complex browser-wise.

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](#), 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 `callbackHow` 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 errorsIn 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:

The first argument of the callback is reserved for an error if it occurs. Then `callback(err)` is called.

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 DoomAt 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:

We load 1.js, then if there's no error...

We load 2.js, then if there's no error...

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.

We'd like to have something better.

Luckily, there are other ways to avoid such pyramids. One of the best ways is to use “promises”, described in the next chapter.

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};  
Object.assign(this.page, { "url": "https://javascript.info/callbacks", "identifier": "V  
callbacks" }); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true;
```

TutorialThe JavaScript languagePromises, async/await{"@context": "https://
schema.org", "@type": "BreadcrumbList", "itemListElement":

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},  
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://  
javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Promises, async/  
await", "item": "https://javascript.info/async"}]
```

August 14, 2022PromiseImagine 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:

A “producing code” that does something and takes time. For instance, some code that loads the data over a network. That's a “singer”.

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”.

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:

The executor is called automatically and immediately (by new Promise).

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 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 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
```

catchIf 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: finallyJust 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:

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.

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
```

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: loadScriptNext, 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. 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.
TasksRe-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);

```

solutionThe 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 promiseThe 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 `promiseRewrite` the `showCircle` function in the solution of the task
Animated circle with 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` as the base.
solution
Open the solution in a sandbox.
Ctrl + ! • Previous lesson
Ctrl + ! Next lesson
Share Tutorial
map
Comments
read this before commenting...
If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
If you can't understand something in the article – please elaborate.
To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)
var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://www.javascript.info/vpromise-basics","identifier":"vpromise-basics"}); };
var DisqusShortname = "javascriptinfo";
var DisqusEnabled = true;
Tutorial
The JavaScript language
Promises, `async/await`
{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},"{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"},"{"@type":"ListItem","position":3,"name":"Promises, `async/await`","item":"https://javascript.info/async"}]}
April 6, 2023
Promises chaining
Let's return to the problem mentioned in the chapter `Introduction: callbacks`: we have a sequence of asynchronous tasks to be performed one after another — for instance, loading scripts.
How can we code it well?
Promises provide a couple of recipes to do that.
In this chapter we cover promise chaining.
It looks like this:

```

    new Promise(function(resolve, reject) {
      setTimeout(() => resolve(1), 1000); // (*)
    }).then(function(result) { // (**)
      alert(result); // 1
      return result * 2;
    }).then(function(result) { // (***)
      alert(result); // 2
      return result * 2;
    }).then(function(result) {
      alert(result); // 4
      return result * 2;
    });

```

The idea is that the result is passed through the chain of .then handlers. Here the flow is:

The initial promise resolves in 1 second (*),
 Then the .then handler is called (**), which in turn creates a new promise (resolved with 2 value).
 The next then (***) gets the result of the previous one, processes it (doubles) and passes it to the next handler.
 ...and so on.

As the result is passed along the chain of handlers, we can see a sequence of alert calls: 1 ! 2 ! 4.

The whole thing works, because every call to a `.then` returns a new promise, so that we can call the next `.then` on it.

When a handler returns a value, it becomes the result of that promise, so the next `.then` is called with it.

A classic newbie error: technically we can also add many `.then` to a single promise. This is not chaining.

For example:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

What we did here is just adding several handlers to one promise. They don't pass the result to each other; instead they process it independently.

Here's the picture (compare it with the chaining above):

All `.then` on the same promise get the same result – the result of that promise. So in the code above all alert show the same: 1.

In practice we rarely need multiple handlers for one promise. Chaining is used much more often.

Returning promisesA handler, used in `.then(handler)` may create and return a promise. In that case further handlers wait until it settles, and then get its result.

For instance:

```
new Promise(function(resolve, reject) {  
  setTimeout(() => resolve(1), 1000);  
}).then(function(result) {  
  alert(result); // 1  
  
  return new Promise((resolve, reject) => { // (*)  
    setTimeout(() => resolve(result * 2), 1000);  
  });  
}).then(function(result) { // (**)  
  alert(result); // 2  
  
  return new Promise((resolve, reject) => {  
    setTimeout(() => resolve(result * 2), 1000);  
  });  
}).then(function(result) {
```

```
    alert(result); // 4
  });
```

Here the first `.then` shows 1 and returns new `Promise(...)` in the line (*). After one second it resolves, and the result (the argument of `resolve`, here it's `result * 2`) is passed on to the handler of the second `.then`. That handler is in the line (**), it shows 2 and does the same thing.

So the output is the same as in the previous example: 1 ! 2 ! 4, but now with 1 second delay between alert calls.

Returning promises allows us to build chains of asynchronous actions.

Example: `loadScript` Let's use this feature with the promisified `loadScript`, defined in the previous chapter, to load scripts one by one, in sequence:

```
    loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // use functions declared in scripts
    // to show that they indeed loaded
    one();
    two();
    three();
  });
```

This code can be made bit shorter with arrow functions:

```

    loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
  });

```

Here each `loadScript` call returns a promise, and the next `.then` runs when it resolves. Then it initiates the loading of the next script. So scripts are loaded one after another.

We can add more asynchronous actions to the chain. Please note that the code is still “flat” — it grows down, not to the right. There are no signs of the “pyramid of doom”. Technically, we could add `.then` directly to each `loadScript`, like this:

```

    loadScript("/article/promise-chaining/one.js").then(script1 => {
    loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
    // this function has access to variables script1, script2 and script3

```

```

    one();
    two();
    three();
  });
});
});

```

This code does the same: loads 3 scripts in sequence. But it “grows to the right”. So we have the same problem as with callbacks.

People who start to use promises sometimes don’t know about chaining, so they write it this way. Generally, chaining is preferred.

Sometimes it’s ok to write `.then` directly, because the nested function has access to the outer scope. In the example above the most nested callback has access to all variables `script1`, `script2`, `script3`. But that’s an exception rather than a rule.

Thenables

To be precise, a handler may return not exactly a promise, but a so-called “thenable” object – an arbitrary object that has a method `.then`. It will be treated the same way as a promise.

The idea is that 3rd-party libraries may implement “promise-compatible” objects of their own. They can have an extended set of methods, but also be compatible with native promises, because they implement `.then`.

Here’s an example of a thenable object:

```

class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after the 1 second
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

```

```

}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // shows 2 after 1000ms

```

JavaScript checks the object returned by the `.then` handler in line (*): if it has a callable method named `then`, then it calls that method providing native functions `resolve`, `reject` as arguments (similar to an executor) and waits until one of them is called. In the example above `resolve(2)` is called after 1 second (**). Then the result is passed further down the chain.

This feature allows us to integrate custom objects with promise chains without having to inherit from `Promise`.

Bigger example: fetchIn frontend programming, promises are often used for network requests. So let's see an extended example of that.

We'll use the `fetch` method to load the information about the user from the remote server. It has a lot of optional parameters covered in separate chapters, but the basic syntax is quite simple:

```
let promise = fetch(url);
```

This makes a network request to the url and returns a promise. The promise resolves with a response object when the remote server responds with headers, but before the full response is downloaded.

To read the full response, we should call the method `response.text()`: it returns a promise that resolves when the full text is downloaded from the remote server, with that text as a result.

The code below makes a request to `user.json` and loads its text from the server:

```

    fetch('/article/promise-chaining/user.json')
// .then below runs when the remote server responds
.then(function(response) {
  // response.text() returns a new promise that resolves with the full response text
  // when it loads
  return response.text();
})
.then(function(text) {
  // ...and here's the content of the remote file
  alert(text); // {"name": "iliakan", "isAdmin": true}
});

```

The response object returned from fetch also includes the method `response.json()` that reads the remote data and parses it as JSON. In our case that's even more convenient, so let's switch to it. We'll also use arrow functions for brevity:

```

// same as above, but response.json() parses the remote content as JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, got user name

```

Now let's do something with the loaded user. For instance, we can make one more request to GitHub, load the user profile and show the avatar:

```

        // Make a request for user.json
fetch('/article/promise-chaining/user.json')
// Load it as json
.then(response => response.json())
// Make a request to GitHub
.then(user => fetch(`https://api.github.com/users/${user.name}`))
// Load the response as json
.then(response => response.json())
// Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe animate it)
.then(githubUser => {
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  setTimeout(() => img.remove(), 3000); // (*)
});

```

The code works; see comments about the details. However, there's a potential problem in it, a typical error for those who begin to use promises. Look at the line (*): how can we do something after the avatar has finished showing and gets removed? For instance, we'd like to show a form for editing that user or something else. As of now, there's no way. To make the chain extendable, we need to return a promise that resolves when the avatar finishes showing. Like this:

```

    fetch('/article/promise-chaining/user.json')
.then(response => response.json())
.then(user => fetch(`https://api.github.com/users/${user.name}`))
.then(response => response.json())
.then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
        img.remove();
        resolve(githubUser); // (**)
    }, 3000);
}))
// triggers after 3 seconds
.then(githubUser => alert(`Finished showing ${githubUser.name}`));

```

That is, the `.then` handler in line (*) now returns new Promise, that becomes settled only after the call of `resolve(githubUser)` in `setTimeout (**)`. The next `.then` in the chain will wait for that.

As a good practice, an asynchronous action should always return a promise. That makes it possible to plan actions after it; even if we don't plan to extend the chain now, we may need it later.

Finally, we can split the code into reusable functions:

```

function loadJson(url) {
return fetch(url)
    .then(response => response.json());
}

```



```

}

function loadGithubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Use them:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
// ...

```

Summary If a `.then` (or `catch/finally`, doesn't matter) handler returns a promise, the rest of the chain waits until it settles. When it does, its result (or error) is passed further. Here's a full picture:

Tasks Promise: then versus catch Are these code fragments equal? In other words, do they behave the same way in any circumstances, for any handler functions?

```
promise.then(f1).catch(f2);
```

Versus:

```
promise.then(f1, f2);
```

solutionThe short answer is: no, they are not equal:
The difference is that if an error happens in f1, then it is handled by .catch here:

```
promise  
.then(f1)  
.catch(f2);
```

...But not here:

```
promise  
.then(f1, f2);
```

That's because an error is passed down the chain, and in the second code piece there's no chain below `f1`.

In other words, `.then` passes results/errors to the next `.then/catch`. So in the first example, there's a catch below, and in the second one there isn't, so the error is unhandled.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/promise-chaining", "identifier": "Vpromise-chaining" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languagePromises, `async/await`{ "@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Promises, `async/await`", "item": "https://javascript.info/async"}]}June 18, 2022Error handling with promisesPromise chains are great at error handling. When a promise rejects, the control jumps to the closest rejection handler. That's very convenient in practice. For instance, in the code below the URL to fetch is wrong (no such site) and `.catch` handles the error:

```
fetch('https://no-such-server.blabla') // rejects  
.then(response => response.json())  
.catch(err => alert(err)) // TypeError: failed to fetch (the text may vary)
```

As you can see, the `.catch` doesn't have to be immediate. It may appear after one or maybe several `.then`.

Or, maybe, everything is all right with the site, but the response is not valid JSON. The easiest way to catch all errors is to append `.catch` to the end of chain:

```
    fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Normally, such `.catch` doesn't trigger at all. But if any of the promises above rejects (a network problem or invalid json or whatever), then it would catch it.

Implicit try...catch The code of a promise executor and promise handlers has an "invisible try..catch" around it. If an exception happens, it gets caught and treated as a rejection.

For instance, this code:

```
    new Promise((resolve, reject) => {  
      throw new Error("Whoops!");  
    }).catch(alert); // Error: Whoops!
```

...Works exactly the same as this:

```
    new Promise((resolve, reject) => {  
      reject(new Error("Whoops!"));  
    }).catch(alert); // Error: Whoops!
```

The "invisible try..catch" around the executor automatically catches the error and turns it into rejected promise.

This happens not only in the executor function, but in its handlers as well. If we throw inside a .then handler, that means a rejected promise, so the control jumps to the nearest error handler.

Here's an example:

```
    new Promise((resolve, reject) => {
      resolve("ok");
    }).then((result) => {
      throw new Error("Whoops!"); // rejects the promise
    }).catch(alert); // Error: Whoops!
```

This happens for all errors, not just those caused by the throw statement. For example, a programming error:

```
    new Promise((resolve, reject) => {
      resolve("ok");
    }).then((result) => {
      blabla(); // no such function
    }).catch(alert); // ReferenceError: blabla is not defined
```

The final `.catch` not only catches explicit rejections, but also accidental errors in the handlers above.

RethrowingAs we already noticed, `.catch` at the end of the chain is similar to `try..catch`. We may have as many `.then` handlers as we want, and then use a single `.catch` at the end to handle errors in all of them.

In a regular `try..catch` we can analyze the error and maybe rethrow it if it can't be handled. The same thing is possible for promises.

If we throw inside `.catch`, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the next closest successful `.then` handler.

In the example below the `.catch` successfully handles the error:

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

Here the `.catch` block finishes normally. So the next successful `.then` handler is called.

In the example below we see the other situation with `.catch`. The handler (*) catches the error and just can't handle it (e.g. it only knows how to handle `URIError`), so it throws it again:

```
// the execution: catch -> catch
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)
```

```

if (error instanceof URIError) {
  // handle it
} else {
  alert("Can't handle such error");

  throw error; // throwing this or another error jumps to the next catch
}

}).then(function() {
  /* doesn't run here */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
  // don't return anything => execution goes the normal way

});

```

The execution jumps from the first `.catch (*)` to the next one `(**)` down the chain. Unhandled rejections What happens when an error is not handled? For instance, we forgot to append `.catch` to the end of the chain, like here:

```

new Promise(function() {
  noSuchFunction(); // Error here (no such function)
})
.then(() => {
  // successful promise handlers, one or more
}); // without .catch at the end!

```

In case of an error, the promise becomes rejected, and the execution should jump to the closest rejection handler. But there is none. So the error gets “stuck”. There’s no code to handle it.

In practice, just like with regular unhandled errors in code, it means that something has gone terribly wrong.

What happens when a regular error occurs and is not caught by `try..catch`? The script dies with a message in the console. A similar thing happens with unhandled promise rejections.

The JavaScript engine tracks such rejections and generates a global error in that case. You can see it in the console if you run the example above.

In the browser we can catch such errors using the event `unhandledrejection`:

```
    window.addEventListener('unhandledrejection', function(event) {  
    // the event object has two special properties:  
    alert(event.promise); // [object Promise] - the promise that generated the error  
    alert(event.reason); // Error: Whoops! - the unhandled error object  
});  
  
new Promise(function() {  
    throw new Error("Whoops!");  
}); // no catch to handle the error
```

The event is the part of the HTML standard.

If an error occurs, and there's no `.catch`, the `unhandledrejection` handler triggers, and gets the event object with the information about the error, so we can do something. Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report the incident to the server.

In non-browser environments like Node.js there are other ways to track unhandled errors.

Summary

`.catch` handles errors in promises of all kinds: be it a `reject()` call, or an error thrown in a handler.

`.then` also catches errors in the same manner, if given the second argument (which is the error handler).

We should place `.catch` exactly in places where we want to handle errors and know how to handle them. The handler should analyze errors (custom error classes help) and

rethrow unknown ones (maybe they are programming mistakes).

It's ok not to use `.catch` at all, if there's no way to recover from an error.

In any case we should have the `unhandledrejection` event handler (for browsers, and analogs for other environments) to track unhandled errors and inform the user (and probably our server) about them, so that our app never "just dies".

TasksError in setTimeoutWhat do you think? Will the `.catch` trigger? Explain your answer.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

solutionThe answer is: no, it won't:

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

As said in the chapter, there's an "implicit `try..catch`" around the function code. So all synchronous errors are handled.

But here the error is generated not while the executor is running, but later. So the promise can't handle it.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before

commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/promise-error-handling", "identifier": "promise-error-handling" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languagePromises, async/await

```
{ "@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{ "@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/" }, { "@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js" }, { "@type": "ListItem", "position": 3, "name": "Promises, async/await", "item": "https://javascript.info/async" } ] }
```

February 20, 2022Promise APIThere are 6 static methods in the Promise class. We'll quickly cover their use cases here.

Promise.allLet's say we want many promises to execute in parallel and wait until all of them are ready.

For instance, download several URLs in parallel and process the content once they are all done.

That's what Promise.all is for.

The syntax is:

```
let promise = Promise.all(iterable);
```

Promise.all takes an iterable (usually, an array of promises) and returns a new promise.

The new promise resolves when all listed promises are resolved, and the array of their results becomes its result.

For instance, the Promise.all below settles after 3 seconds, and then its result is an array [1, 2, 3]:

```

    Promise.all([
      new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
      new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
      new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
    ]).then(alert); // 1,2,3 when promises are ready: each promise contributes an array member

```

Please note that the order of the resulting array members is the same as in its source promises. Even though the first promise takes the longest time to resolve, it's still first in the array of results.

A common trick is to map an array of job data into an array of promises, and then wrap that into `Promise.all`.

For instance, if we have an array of URLs, we can fetch them all like this:

```

    let urls = [
      'https://api.github.com/users/iliakan',
      'https://api.github.com/users/remy',
      'https://api.github.com/users/jeresig'
    ];

    // map every url to the promise of the fetch
    let requests = urls.map(url => fetch(url));

    // Promise.all waits until all jobs are resolved
    Promise.all(requests)
      .then(responses => responses.forEach(
        response => alert(`${response.url}: ${response.status}`)
      ));

```

A bigger example with fetching user information for an array of GitHub users by

their names (we could fetch an array of goods by their ids, the logic is identical):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // all responses are resolved successfully
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // shows 200 for every url
    }

    return responses;
  })
  // map array of responses into an array of response.json() to read their content
  .then(responses => Promise.all(responses.map(r => r.json())))
  // all JSON answers are parsed: "users" is the array of them
  .then(users => users.forEach(user => alert(user.name)));
```

If any of the promises is rejected, the promise returned by Promise.all immediately rejects with that error.
For instance:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")),
    2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Here the second promise rejects in two seconds. That leads to an immediate rejection of `Promise.all`, so `.catch` executes: the rejection error becomes the outcome of the entire `Promise.all`.

In case of an error, other promises are ignored

If one promise rejects, `Promise.all` immediately rejects, completely forgetting about the other ones in the list. Their results are ignored.

For example, if there are multiple fetch calls, like in the example above, and one fails, the others will still continue to execute, but `Promise.all` won't watch them anymore. They will probably settle, but their results will be ignored.

`Promise.all` does nothing to cancel them, as there's no concept of "cancellation" in promises. In another chapter we'll cover `AbortController` that can help with that, but it's not a part of the Promise API.

`Promise.all(iterable)` allows non-promise "regular" values in iterable

Normally, `Promise.all(...)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is".

For instance, here the results are `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
```

```
    }),  
    2,  
    3  
  ]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to `Promise.all` where convenient.

`Promise.allSettled`

A recent addition

This is a recent addition to the language.
Old browsers may need polyfills.

`Promise.all` rejects as a whole if any promise rejects. That's good for "all or nothing" cases, when we need all results successful to proceed:

```
    Promise.all([  
      fetch('/template.html'),  
      fetch('/style.css'),  
      fetch('/data.json')  
    ]).then(render); // render method needs results of all fetches
```

`Promise.allSettled` just waits for all promises to settle, regardless of the result. The resulting array has:

`{status:"fulfilled", value:result}` for successful responses,
`{status:"rejected", reason:error}` for errors.

For example, we'd like to fetch the information about multiple users. Even if one request fails, we're still interested in the others.

Let's use `Promise.allSettled`:

```

    let urls = [
      'https://api.github.com/users/iliakan',
      'https://api.github.com/users/remy',
      'https://no-such-url'
    ];

    Promise.allSettled(urls.map(url => fetch(url)))
      .then(results => { // (*)
        results.forEach((result, num) => {
          if (result.status == "fulfilled") {
            alert(`${urls[num]}: ${result.value.status}`);
          }
          if (result.status == "rejected") {
            alert(`${urls[num]}: ${result.reason}`);
          }
        });
      });

```

The results in the line (*) above will be:

```

[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]

```

So for each promise we get its status and value/error.
 Polyfill If the browser doesn't support Promise.allSettled, it's easy to polyfill:

```

if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

```



```

const resolveHandler = value => ({ status: 'fulfilled', value });

Promise.allSettled = function (promises) {
  const convertedPromises = promises.map(p =>
Promise.resolve(p).then(resolveHandler, rejectHandler));
  return Promise.all(convertedPromises);
};
}

```

In this code, `promises.map` takes input values, turns them into promises (just in case a non-promise was passed) with `p => Promise.resolve(p)`, and then adds `.then` handler to every one.

That handler turns a successful result value into `{status:'fulfilled', value}`, and an error reason into `{status:'rejected', reason}`. That's exactly the format of `Promise.allSettled`. Now we can use `Promise.allSettled` to get the results of all given promises, even if some of them reject.

`Promise.race` Similar to `Promise.all`, but waits only for the first settled promise and gets its result (or error).

The syntax is:

```
let promise = Promise.race(iterable);
```

For instance, here the result will be 1:

```

Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")),
2000)),

```

```
new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, so it became the result. After the first settled promise “wins the race”, all further results/errors are ignored. Promise.any Similar to Promise.race, but waits only for the first fulfilled promise and gets its result. If all of the given promises are rejected, then the returned promise is rejected with AggregateError – a special error object that stores all promise errors in its errors property.

The syntax is:

```
let promise = Promise.any(iterable);
```

For instance, here the result will be 1:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")),
    1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, but it was rejected, so the second promise became the result. After the first fulfilled promise “wins the race”, all further results are ignored.

Here’s an example when all promises fail:

```

    Promise.any([
      new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),
      new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))
    ]).catch(error => {
      console.log(error.constructor.name); // AggregateError
      console.log(error.errors[0]); // Error: Ouch!
      console.log(error.errors[1]); // Error: Error!
    });

```

As you can see, error objects for failed promises are available in the errors property of the AggregateError object.

Promise.resolve/rejectMethods Promise.resolve and Promise.reject are rarely needed in modern code, because async/await syntax (we'll cover it a bit later) makes them somewhat obsolete.

We cover them here for completeness and for those who can't use async/await for some reason.

Promise.resolvePromise.resolve(value) creates a resolved promise with the result value. Same as:

```
let promise = new Promise(resolve => resolve(value));
```

The method is used for compatibility, when a function is expected to return a promise.

For example, the loadCached function below fetches a URL and remembers (caches) its content. For future calls with the same URL it immediately gets the previous content from cache, but uses Promise.resolve to make a promise of it, so the returned value is always a promise:

```

    let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}

```

We can write `loadCached(url).then(...)`, because the function is guaranteed to return a promise. We can always use `.then` after `loadCached`. That's the purpose of `Promise.resolve` in the line `(*)`.
`Promise.rejectPromise.reject(error)` creates a rejected promise with error.
 Same as:

```
let promise = new Promise((resolve, reject) => reject(error));
```

In practice, this method is almost never used.
 Summary There are 6 static methods of Promise class:

`Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, it becomes the error of `Promise.all`, and all other results are ignored.

`Promise.allSettled(promises)` (recently added method) – waits for all promises to settle and returns their results as an array of objects with:

status: "fulfilled" or "rejected"
 value (if fulfilled) or reason (if rejected).

Promise.race(promises) – waits for the first promise to settle, and its result/error becomes the outcome.

Promise.any(promises) (recently added method) – waits for the first promise to fulfill, and its result becomes the outcome. If all of the given promises are rejected, AggregateError becomes the error of Promise.any.

Promise.resolve(value) – makes a resolved promise with the given value.

Promise.reject(error) – makes a rejected promise with the given error.

Of all these, Promise.all is probably the most common in practice.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, { "url": "https://javascript.info/promise-api", "identifier": "Vpromise-api" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languagePromises, async/await{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":

[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},

{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://

javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Promises, async/

await", "item": "https://javascript.info/async"}]]October 18,

2022Promisification“Promisification” is a long word for a simple transformation. It’s the conversion of a function that accepts a callback into a function that returns a promise.

Such transformations are often required in real-life, as many functions and libraries are callback-based. But promises are more convenient, so it makes sense to promisify them.

For better understanding, let’s see an example.

For instance, we have loadScript(src, callback) from the chapter Introduction: callbacks.

```
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);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})

```

The function loads a script with the given src, and then calls callback(err) in case of an error, or callback(null, script) in case of successful loading. That's a widespread agreement for using callbacks, we saw it before.

Let's promisify it.

We'll make a new function loadScriptPromise(src), that does the same (loads the script), but returns a promise instead of using callbacks.

In other words, we pass it only src (no callback) and get a promise in return, that resolves with script when the load is successful, and rejects with the error otherwise. Here it is:

```

let loadScriptPromise = function(src) {
return new Promise((resolve, reject) => {
  loadScript(src, (err, script) => {
    if (err) reject(err);
    else resolve(script);
  });
});
};

// usage:
// loadScriptPromise('path/script.js').then(...)

```

As we can see, the new function is a wrapper around the original loadScript function. It calls it providing its own callback that translates to promise resolve/reject. Now loadScriptPromise fits well in promise-based code. If we like promises more than callbacks (and soon we'll see more reasons for that), then we will use it instead. In practice we may need to promisify more than one function, so it makes sense to use a helper.

We'll call it promisify(f): it accepts a to-promisify function f and returns a wrapper function.

```

    function promisify(f) {
return function (...args) { // return a wrapper-function (*)
    return new Promise((resolve, reject) => {
        function callback(err, result) { // our custom callback for f (**)
            if (err) {
                reject(err);
            } else {
                resolve(result);
            }
        }
    })

    args.push(callback); // append our custom callback to the end of f arguments

    f.call(this, ...args); // call the original function
  });
};
}

// usage:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);

```

The code may look a bit complex, but it's essentially the same that we wrote above, while promisifying loadScript function.

A call to `promisify(f)` returns a wrapper around `f (*)`. That wrapper returns a promise and forwards the call to the original `f`, tracking the result in the custom callback (**).

Here, `promisify` assumes that the original function expects a callback with exactly two arguments (`err, result`). That's what we encounter most often. Then our custom callback is in exactly the right format, and `promisify` works great for such a case.

But what if the original `f` expects a callback with more arguments `callback(err, res1, res2, ...)`?

We can improve our helper. Let's make a more advanced version of `promisify`.

When called as `promisify(f)` it should work similar to the version above.

When called as `promisify(f, true)`, it should return the promise that resolves with the array of callback results. That's exactly for callbacks with many arguments.

```

    // promisify(f, true) to get array of results
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // our custom callback for f
        if (err) {
          reject(err);
        } else {
          // resolve with all callback results if manyArgs is specified
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// usage:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);

```

As you can see it's essentially the same as above, but resolve is called with only one or all arguments depending on whether manyArgs is truthy. For more exotic callback formats, like those without err at all: callback(result), we can promisify such functions manually without using the helper. There are also modules with a bit more flexible promisification functions, e.g. es6-promisify. In Node.js, there's a built-in util.promisify function for that.

Please note:

Promisification is a great approach, especially when you use async/await (covered later in the chapter Async/await), but not a total replacement for callbacks. Remember, a promise may have only one result, but a callback may technically be called many times. So promisification is only meant for functions that call the callback once. Further calls will be ignored.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines –

wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/promisify", "identifier": "promisify" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript languagePromises, async/await{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Promises, async/await", "item": "https://javascript.info/async"}]}December 12, 2021MicrotasksPromise handlers .then/.catch/.finally are always asynchronous. Even when a Promise is immediately resolved, the code on the lines below .then/.catch/.finally will still execute before these handlers. Here's a demo:

```
let promise = Promise.resolve();

promise.then(() => alert("promise done!"));

alert("code finished"); // this alert shows first
```

If you run it, you see code finished first, and then promise done!. That's strange, because the promise is definitely done from the beginning. Why did the .then trigger afterwards? What's going on? Microtasks queueAsynchronous tasks need proper management. For that, the ECMA standard specifies an internal queue PromiseJobs, more often referred to as the “microtask queue” (V8 term). As stated in the specification:

The queue is first-in-first-out: tasks enqueued first are run first. Execution of a task is initiated only when nothing else is running.

Or, to put it more simply, when a promise is ready, its .then/catch/finally handlers are

put into the queue; they are not executed yet. When the JavaScript engine becomes free from the current code, it takes a task from the queue and executes it. That's why "code finished" in the example above shows first.

Promise handlers always go through this internal queue. If there's a chain with multiple `.then/catch/finally`, then every one of them is executed asynchronously. That is, it first gets queued, then executed when the current code is complete and previously queued handlers are finished. What if the order matters for us? How can we make code finished appear after promise done? Easy, just put it into the queue with `.then`:

```
Promise.resolve()  
  .then(() => alert("promise done!"))  
  .then(() => alert("code finished"));
```

Now the order is as intended. **Unhandled rejection** Remember the unhandledrejection event from the article Error handling with promises? Now we can see exactly how JavaScript finds out that there was an unhandled rejection. An "unhandled rejection" occurs when a promise error is not handled at the end of the microtask queue. Normally, if we expect an error, we add `.catch` to the promise chain to handle it:

```
let promise = Promise.reject(new Error("Promise Failed!"));
promise.catch(err => alert('caught'));

// doesn't run: error handled
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

But if we forget to add `.catch`, then, after the microtask queue is empty, the engine triggers the event:

```
let promise = Promise.reject(new Error("Promise Failed!"));

// Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

What if we handle the error later? Like this:

```

    let promise = Promise.reject(new Error("Promise Failed!"));
    setTimeout(() => promise.catch(err => alert('caught')), 1000);

// Error: Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));

```

Now, if we run it, we'll see Promise Failed! first and then caught. If we didn't know about the microtasks queue, we could wonder: "Why did unhandledrejection handler run? We did catch and handle the error!" But now we understand that unhandledrejection is generated when the microtask queue is complete: the engine examines promises and, if any of them is in the "rejected" state, then the event triggers. In the example above, .catch added by setTimeout also triggers. But it does so later, after unhandledrejection has already occurred, so it doesn't change anything. Summary Promise handling is always asynchronous, as all promise actions pass through the internal "promise jobs" queue, also called "microtask queue" (V8 term). So .then/catch/finally handlers are always called after the current code is finished. If we need to guarantee that a piece of code is executed after .then/catch/finally, we can add it into a chained .then call.

In most Javascript engines, including browsers and Node.js, the concept of microtasks is closely tied with the "event loop" and "macrotasks". As these have no direct relation to promises, they are covered in another part of the tutorial, in the article Event loop: microtasks and macrotasks.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, {"url":"https://javascript.info/microtask-queue","identifier":"Vmicrotask-queue"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript languagePromises, async/await{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}], [{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Promises, async/await","item":"https://javascript.info/async"}]}

February 6, 2022Async/awaitThere's a special syntax to work with promises in a more comfortable fashion, called "async/await". It's surprisingly easy to understand and use.

Async functionsLet's start with the async keyword. It can be placed before a function, like this:

```
    async function f() {  
    return 1;  
}
```

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically. For instance, this function returns a resolved promise with the result of 1; let's test it:

```
    async function f() {  
    return 1;  
}  
  
f().then(alert); // 1
```

...We could explicitly return a promise, which would be the same:

```
    async function f() {  
    return Promise.resolve(1);  
    }
```

```
f().then(alert); // 1
```

So, `async` ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There's another keyword, `await`, that works only inside `async` functions, and it's pretty cool.

AwaitThe syntax:

```
    // works only inside async functions  
let value = await promise;
```

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

Here's an example with a promise that resolves in 1 second:

```
    async function f() {  
  
    let promise = new Promise((resolve, reject) => {  
        setTimeout(() => resolve("done!"), 1000)  
    });  
  
    let result = await promise; // wait until the promise resolves (*)  
  
    alert(result); // "done!"  
    }
```

```
f();
```

The function execution “pauses” at the line (*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in one second. Let’s emphasize: `await` literally suspends the function execution until the promise settles, and then resumes it with the promise result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc. It’s just a more elegant syntax of getting the promise result than `promise.then`. And, it’s easier to read and write.

Can’t use `await` in regular functions

If we try to use `await` in a non-async function, there would be a syntax error:

```
function f() {  
  let promise = Promise.resolve(1);  
  let result = await promise; // Syntax error  
}
```

We may get this error if we forget to put `async` before a function. As stated earlier, `await` only works inside an async function.

Let’s take the `showAvatar()` example from the chapter Promises chaining and rewrite it using `async/await`:

We’ll need to replace `.then` calls with `await`.

Also we should make the function `async` for them to work.

```

    async function showAvatar() {

// read our JSON
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

// read github user
let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
let githubUser = await githubResponse.json();

// show the avatar
let img = document.createElement('img');
img.src = githubUser.avatar_url;
img.className = "promise-avatar-example";
document.body.append(img);

// wait 3 seconds
await new Promise((resolve, reject) => setTimeout(resolve, 3000));

img.remove();

return githubUser;
}

showAvatar();

```

Pretty clean and easy to read, right? Much better than before.

Modern browsers allow top-level await in modules

In modern browsers, await on top level works just fine, when we're inside a module. We'll cover modules in article Modules, introduction.
For instance:


```
// we assume this code runs at top level, inside a module
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

console.log(user);
```

If we're not using modules, or older browsers must be supported, there's a universal recipe: wrapping into an anonymous async function.
Like this:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

await accepts “thenables”

Like `promise.then`, `await` allows us to use thenable objects (those with a callable `then` method). The idea is that a third-party object may not be a promise, but promise-compatible: if it supports `.then`, that's enough to use it with `await`. Here's a demo Thenable class; the `await` below accepts its instances:

```

    class Thenable {
    constructor(num) {
      this.num = num;
    }
    then(resolve, reject) {
      alert(resolve);
      // resolve with this.num*2 after 1000ms
      setTimeout(() => resolve(this.num * 2), 1000); // (*)
    }
  }

  async function f() {
    // waits for 1 second, then result becomes 2
    let result = await new Thenable(1);
    alert(result);
  }

  f();

```

If await gets a non-promise object with .then, it calls that method providing the built-in functions resolve and reject as arguments (just as it does for a regular Promise executor). Then await waits until one of them is called (in the example above it happens in the line (*)) and then proceeds with the result.

Async class methods

To declare an async class method, just prepend it with async:

```

    class Waiter {
    async wait() {
      return await Promise.resolve(1);
    }
  }

  new Waiter()
    .wait()
    .then(alert); // 1 (this is the same as (result => alert(result)))

```

The meaning is the same: it ensures that the returned value is a promise and enables await.

Error handling If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line.

This code:

```

    async function f() {
    await Promise.reject(new Error("Whoops!"));
  }

```

...is the same as this:

```

    async function f() {
    throw new Error("Whoops!");
  }

```

In real situations, the promise may take some time before it rejects. In that case there will be a delay before await throws an error.

We can catch that error using try..catch, the same way as a regular throw:

```
    async function f() {  
  
    try {  
      let response = await fetch('http://no-such-url');  
    } catch(err) {  
      alert(err); // TypeError: failed to fetch  
    }  
  }  
  
  f();
```

In the case of an error, the control jumps to the catch block. We can also wrap multiple lines:

```
    async function f() {  
  
    try {  
      let response = await fetch('/no-user-here');  
      let user = await response.json();  
    } catch(err) {  
      // catches errors both in fetch and response.json  
      alert(err);  
    }  
  }
```

```
}  
  
f();
```

If we don't have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
async function f() {  
  let response = await fetch('http://no-such-url');  
}  
  
// f() becomes a rejected promise  
f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add `.catch` there, then we get an unhandled promise error (viewable in the console). We can catch such errors using a global `unhandledrejection` event handler as described in the chapter [Error handling with promises](#).

async/await and promise.then/catch

When we use `async/await`, we rarely need `.then`, because `await` handles the waiting for us. And we can use a regular `try..catch` instead of `.catch`. That's usually (but not always) more convenient.

But at the top level of the code, when we're outside any `async` function, we're syntactically unable to use `await`, so it's a normal practice to add `.then/catch` to handle the final result or falling-through error, like in the line `(*)` of the example above.

async/await works well with Promise.all

When we need to wait for multiple promises, we can wrap them in `Promise.all` and then `await`:

```
        // wait for the array of results
let results = await Promise.all([
    fetch(url1),
    fetch(url2),
    ...
]);
```

In the case of an error, it propagates as usual, from the failed promise to `Promise.all`, and then becomes an exception that we can catch using `try..catch` around the call.

Summary The `async` keyword before a function has two effects:

- Makes it always return a promise.
- Allows `await` to be used in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

- If it's an error, an exception is generated — same as if `throw error` were called at that very place.
- Otherwise, it returns the result.

Together they provide a great framework to write asynchronous code that is easy to both read and write.

With `async/await` we rarely need to write `promise.then/catch`, but we still shouldn't forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also `Promise.all` is nice when we are waiting for many tasks simultaneously.

Tasks Rewrite using `async/await` Rewrite this example code from the chapter Promises chaining using `async/await` instead of `.then/catch`:

```

    function loadJson(url) {
return fetch(url)
    .then(response => {
      if (response.status === 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('https://javascript.info/no-such-user.json')
  .catch(alert); // Error: 404

```

solutionThe notes are below the code:

```

    async function loadJson(url) { // (1)
let response = await fetch(url); // (2)

if (response.status === 200) {
  let json = await response.json(); // (3)
  return json;
}

throw new Error(response.status);
}

loadJson('https://javascript.info/no-such-user.json')
  .catch(alert); // Error: 404 (4)

```

Notes:

The function `loadJson` becomes `async`.

All `.then` inside are replaced with `await`.

We can return `response.json()` instead of awaiting for it, like this:

```
    if (response.status === 200) {  
    return response.json(); // (3)  
    }
```

Then the outer code would have to await for that promise to resolve. In our case it doesn't matter.

The error thrown from `loadJson` is handled by `.catch`. We can't use `await loadJson(...)` there, because we're not in an `async` function.

Rewrite "rethrow" with `async/await` Below you can find the "rethrow" example. Rewrite it using `async/await` instead of `.then/catch`.
And get rid of the recursion in favour of a loop in `demoGithubUser`: with `async/await` that becomes easy to do.

```
class HttpError extends Error {
```



```

    constructor(response) {
      super(` ${response.status} for ${response.url}`);
      this.name = 'HttpError';
      this.response = response;
    }
  }

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status === 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    });
}

// Ask for a user name until github returns a valid user
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status === 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();

```

solution There are no tricks here. Just replace `.catch` with `try..catch` inside `demoGithubUser` and add `async/await` where needed:

```

    class HttpError extends Error {
    constructor(response) {
        super(`${response.status} for ${response.url}`);
        this.name = 'HttpError';
        this.response = response;
    }
}

```

```

async function loadJson(url) {
    let response = await fetch(url);
    if (response.status == 200) {
        return response.json();
    } else {
        throw new HttpError(response);
    }
}

```

```

// Ask for a user name until github returns a valid user
async function demoGithubUser() {

```

```

    let user;
    while(true) {
        let name = prompt("Enter a name?", "iliakan");

        try {
            user = await loadJson(`https://api.github.com/users/${name}`);
            break; // no error, exit loop
        } catch(err) {
            if (err instanceof HttpError && err.response.status == 404) {
                // loop continues after the alert
                alert("No such user, please reenter.");
            } else {
                // unknown error, rethrow
                throw err;
            }
        }
    }
}

```

```
    alert(`Full name: ${user.name}.`);  
    return user;  
}
```

```
demoGithubUser();
```

Call async from non-async We have a “regular” function called f. How can you call the async function wait() and use its result inside of f?

```
    async function wait() {  
        await new Promise(resolve => setTimeout(resolve, 1000));  
  
        return 10;  
    }  
  
function f() {  
    // ...what should you write here?  
    // we need to call async wait() and wait to get 10  
    // remember, we can't use "await"  
}
```

P.S. The task is technically very simple, but the question is quite common for developers new to async/await.
solution That's the case when knowing how it works inside is helpful.
Just treat async call as promise and attach .then to it:

```

    async function wait() {
      await new Promise(resolve => setTimeout(resolve, 1000));

      return 10;
    }

    function f() {
      // shows 10 after 1 second
      wait().then(result => alert(result));
    }

    f();

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/async-await", "identifier": "Vasync-await" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;TutorialThe JavaScript language{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}]}Generators, advanced iterationGeneratorsAsync iteration and generatorsCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapTutorialThe JavaScript languageGenerators, advanced iteration{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Generators, advanced iteration", "item": "https://javascript.info/generators-iterators"}]}August 30, 2022GeneratorsRegular functions return only one, single value (or nothing). Generators can return (“yield”) multiple values, one after another, on-demand. They work great with iterables, allowing to create data streams with ease. Generator functionsTo create a generator, we need a special syntax construct: function*, so-called “generator function”. It looks like this:

```

function* generateSequence() {

```

```
yield 1;  
yield 2;  
return 3;  
}
```

Generator functions behave differently from regular ones. When such function is called, it doesn't run its code. Instead it returns a special object, called "generator object", to manage the execution. Here, take a look:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
// "generator function" creates "generator object"  
let generator = generateSequence();  
alert(generator); // [object Generator]
```

The function code execution hasn't started yet:

The main method of a generator is `next()`. When called, it runs the execution until the nearest `yield <value>` statement (value can be omitted, then it's undefined). Then the function execution pauses, and the yielded value is returned to the outer code. The result of `next()` is always an object with two properties:

value: the yielded value.

done: true if the function code has finished, otherwise false.

For instance, here we create the generator and get its first yielded value:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
let generator = generateSequence();  
  
let one = generator.next();  
  
alert(JSON.stringify(one)); // {value: 1, done: false}
```

As of now, we got the first value only, and the function execution is on the second line:

Let's call `generator.next()` again. It resumes the code execution and returns the next yield:

```
let two = generator.next();  
  
alert(JSON.stringify(two)); // {value: 2, done: false}
```

And, if we call it a third time, the execution reaches the return statement that finishes the function:

```
let three = generator.next();  
alert(JSON.stringify(three)); // {value: 3, done: true}
```

Now the generator is done. We should see it from `done:true` and process `value:3` as the final result.
New calls to `generator.next()` don't make sense any more. If we do them, they return the same object: `{done: true}`.

`function* f(...)` or `function *f(...)`?

Both syntaxes are correct.

But usually the first syntax is preferred, as the star `*` denotes that it's a generator function, it describes the kind, not the name, so it should stick with the function keyword.

Generators are iterableAs you probably already guessed looking at the `next()` method, generators are iterable.

We can loop over their values using `for..of`:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}
```

```
let generator = generateSequence();
```

```
for(let value of generator) {  
  alert(value); // 1, then 2  
}
```

Looks a lot nicer than calling `.next().value`, right?
...But please note: the example above shows 1, then 2, and that's all. It doesn't show 3!
It's because `for..of` iteration ignores the last value, when `done: true`. So, if we want all results to be shown by `for..of`, we must return them with `yield`:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  yield 3;  
}
```

```
let generator = generateSequence();
```

```
for(let value of generator) {  
  alert(value); // 1, then 2, then 3  
}
```


As generators are iterable, we can call all related functionality, e.g. the spread syntax ...:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
let sequence = [0, ...generateSequence()];  
  
alert(sequence); // 0, 1, 2, 3
```

In the code above, ...generateSequence() turns the iterable generator object into an array of items (read more about the spread syntax in the chapter Rest parameters and spread syntax)

Using generators for iterablesSome time ago, in the chapter Iterables we created an iterable range object that returns values from..to.

Here, let's remember the code:

```

    let range = {
from: 1,
to: 5,

// for..of range calls this method once in the very beginning
[Symbol.iterator]() {
  // ...it returns the iterator object:
  // onward, for..of works only with that object, asking it for next values
  return {
    current: this.from,
    last: this.to,

    // next() is called on each iteration by the for..of loop
    next() {
      // it should return the value as an object {done:..., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
}

// iteration over range returns numbers from range.from to range.to
alert([...range]); // 1,2,3,4,5

```

We can use a generator function for iteration by providing it as `Symbol.iterator`. Here's the same range, but much more compact:

```

    let range = {
from: 1,

```

to: 5,

```
*[Symbol.iterator]() { // a shorthand for [Symbol.iterator]: function*()
  for(let value = this.from; value <= this.to; value++) {
    yield value;
  }
}
};

alert( [...range] ); // 1,2,3,4,5
```

That works, because `range[Symbol.iterator]()` now returns a generator, and generator methods are exactly what `for..of` expects:

it has a `.next()` method
that returns values in the form `{value: ..., done: true/false}`

That's not a coincidence, of course. Generators were added to JavaScript language with iterators in mind, to implement them easily.

The variant with a generator is much more concise than the original iterable code of `range`, and keeps the same functionality.

Generators may generate values forever

In the examples above we generated finite sequences, but we can also make a generator that yields values forever. For instance, an unending sequence of pseudo-random numbers.

That surely would require a `break` (or `return`) in `for..of` over such generator. Otherwise, the loop would repeat forever and hang.

Generator composition Generator composition is a special feature of generators that allows to transparently “embed” generators in each other.

For instance, we have a function that generates a sequence of numbers:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

Now we'd like to reuse it to generate a more complex sequence:

first, digits 0..9 (with character codes 48...57),
followed by uppercase alphabet letters A..Z (character codes 65...90)
followed by lowercase alphabet letters a..z (character codes 97...122)

We can use this sequence e.g. to create passwords by selecting characters from it (could add syntax characters as well), but let's generate it first.

In a regular function, to combine results from multiple other functions, we call them, store the results, and then join at the end.

For generators, there's a special `yield*` syntax to "embed" (compose) one generator into another.

The composed generator:

```
function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) yield i;  
}
```

```
function* generatePasswordCodes() {  
  
  // 0..9  
  yield* generateSequence(48, 57);  
  
  // A..Z  
  yield* generateSequence(65, 90);  
  
  // a..z  
  yield* generateSequence(97, 122);  
  
}
```

```
let str = "";
```

```
for(let code of generatePasswordCodes()) {  
  str += String.fromCharCode(code);  
}
```

```
alert(str); // 0..9A..Za..z
```

The `yield*` directive delegates the execution to another generator. This term means that `yield* gen` iterates over the generator `gen` and transparently forwards its yields outside. As if the values were yielded by the outer generator. The result is the same as if we inlined the code from nested generators:

```
function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) yield i;  
}
```

```
function* generateAlphaNum() {  
  
  // yield* generateSequence(48, 57);  
  for (let i = 48; i <= 57; i++) yield i;  
  
  // yield* generateSequence(65, 90);  
  for (let i = 65; i <= 90; i++) yield i;  
  
  // yield* generateSequence(97, 122);  
  for (let i = 97; i <= 122; i++) yield i;  
  
}
```

```
let str = '';
```

```
for(let code of generateAlphaNum()) {  
  str += String.fromCharCode(code);  
}
```

```
alert(str); // 0..9A..Za..z
```

A generator composition is a natural way to insert a flow of one generator into another. It doesn't use extra memory to store intermediate results.

"yield" is a two-way street Until this moment, generators were similar to iterable objects, with a special syntax to generate values. But in fact they are much more powerful and flexible.

That's because yield is a two-way street: it not only returns the result to the outside, but also can pass the value inside the generator.

To do so, we should call `generator.next(arg)`, with an argument. That argument becomes the result of `yield`.

Let's see an example:

```
function* gen() {  
  // Pass a question to the outer code and wait for an answer  
  let result = yield "2 + 2 = ?"; // (*)  
  
  alert(result);  
}  
  
let generator = gen();  
  
let question = generator.next().value; // <-- yield returns the value  
  
generator.next(4); // --> pass the result into the generator
```

The first call `generator.next()` should be always made without an argument (the argument is ignored if passed). It starts the execution and returns the result of the first

yield "2+2=?". At this point the generator pauses the execution, while staying on the line (*).

Then, as shown at the picture above, the result of yield gets into the question variable in the calling code.

On generator.next(4), the generator resumes, and 4 gets in as the result: let result = 4.

Please note, the outer code does not have to immediately call next(4). It may take time. That's not a problem: the generator will wait.

For instance:

```
// resume the generator after some time
setTimeout(() => generator.next(4), 1000);
```

As we can see, unlike regular functions, a generator and the calling code can exchange results by passing values in next/yield.

To make things more obvious, here's another example, with more calls:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"
```

```
alert( generator.next(4).value ); // "3 * 3 = ?"
```

```
alert( generator.next(9).done ); // true
```

The execution picture:

The first `.next()` starts the execution... It reaches the first `yield`.

The result is returned to the outer code.

The second `.next(4)` passes 4 back to the generator as the result of the first `yield`, and resumes the execution.

...It reaches the second `yield`, that becomes the result of the generator call.

The third `next(9)` passes 9 into the generator as the result of the second `yield` and resumes the execution that reaches the end of the function, so `done: true`.

It's like a "ping-pong" game. Each `next(value)` (excluding the first one) passes a value into the generator, that becomes the result of the current `yield`, and then gets back the result of the next `yield`.

`generator.throw` As we observed in the examples above, the outer code may pass a value into the generator, as the result of `yield`.

...But it can also initiate (`throw`) an error there. That's natural, as an error is a kind of result.

To pass an error into a `yield`, we should call `generator.throw(err)`. In that case, the `err` is thrown in the line with that `yield`.

For instance, here the `yield` of `"2 + 2 = ?"` leads to an error:

```
function* gen() {  
  try {
```



```

    let result = yield "2 + 2 = ?"; // (1)

    alert("The execution does not reach here, because the exception is thrown above");
  } catch(e) {
    alert(e); // shows the error
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("The answer is not found in my database")); // (2)

```

The error, thrown into the generator at line (2) leads to an exception in line (1) with yield. In the example above, try..catch catches it and shows it.

If we don't catch it, then just like any exception, it "falls out" the generator into the calling code.

The current line of the calling code is the line with generator.throw, labelled as (2). So we can catch it here, like this:

```

    function* generate() {
      let result = yield "2 + 2 = ?"; // Error in this line
    }

let generator = generate();

let question = generator.next().value;

try {
  generator.throw(new Error("The answer is not found in my database"));
} catch(e) {
  alert(e); // shows the error
}

```

```
}
```

If we don't catch the error there, then, as usual, it falls through to the outer calling code (if any) and, if uncaught, kills the script.
generator.return(generator.return(value)) finishes the generator execution and return the given value.

```
function* gen() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const g = gen();  
  
g.next();    // { value: 1, done: false }  
g.return('foo'); // { value: "foo", done: true }  
g.next();    // { value: undefined, done: true }
```

If we again use generator.return() in a completed generator, it will return that value again (MDN).

Often we don't use it, as most of time we want to get all returning values, but it can be useful when we want to stop generator in a specific condition.

Summary

Generators are created by generator functions function* f(...) {...}.

Inside generators (only) there exists a yield operator.

The outer code and the generator may exchange results via next/yield calls.

In modern JavaScript, generators are rarely used. But sometimes they come in handy, because the ability of a function to exchange data with the calling code during the execution is quite unique. And, surely, they are great for making iterable objects. Also, in the next chapter we'll learn async generators, which are used to read streams of asynchronously generated data (e.g paginated fetches over a network) in for await ... of loops.

In web-programming we often work with streamed data, so that's another very important use case.

TasksPseudo-random generatorThere are many areas where we need random data.

One of them is testing. We may need random data: text, numbers, etc. to test things out well.

In JavaScript, we could use `Math.random()`. But if something goes wrong, we'd like to be able to repeat the test, using exactly the same data.

For that, so called "seeded pseudo-random generators" are used. They take a "seed", the first value, and then generate the next ones using a formula so that the same seed yields the same sequence, and hence the whole flow is easily reproducible. We only need to remember the seed to repeat it.

An example of such formula, that generates somewhat uniformly distributed values:

$$\text{next} = \text{previous} * 16807 \% 2147483647$$

If we use 1 as the seed, the values will be:

16807
282475249
1622650073
...and so on...

The task is to create a generator function `pseudoRandom(seed)` that takes seed and creates the generator with this formula.

Usage example:

```
let generator = pseudoRandom(1);  
  
alert(generator.next().value); // 16807  
alert(generator.next().value); // 282475249  
alert(generator.next().value); // 1622650073
```

Open a sandbox with tests.solution

```

    function* pseudoRandom(seed) {
    let value = seed;

    while(true) {
        value = value * 16807 % 2147483647;
        yield value;
    }

};

let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073

```

Please note, the same can be done with a regular function, like this:

```

    function pseudoRandom(seed) {
    let value = seed;

    return function() {
        value = value * 16807 % 2147483647;
        return value;
    }
}

let generator = pseudoRandom(1);

alert(generator()); // 16807

```

```
alert(generator()); // 282475249
alert(generator()); // 1622650073
```

That also works. But then we lose ability to iterate with `for..of` and to use generator composition, that may be useful elsewhere.

Open the solution with tests in a sandbox. [Ctrl + !•Previous lesson](#) [Ctrl + !Next lesson](#) [Share Tutorial](#) [mapComments](#) [read this before commenting...](#) If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting. If you can't understand something in the article – please elaborate. To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...) `var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://\njavascript.info\ngenerators","identifier":"\ngenerators"}); }; var disqus_shortcode = "javascriptinfo"; var disqus_enabled = true;` [Tutorial](#) [The JavaScript language](#) [Generators, advanced iteration](#) `{"@context":"https://\nschema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Generators, advanced iteration","item":"https://javascript.info/generators-iterators"}]}` [February 4, 2022](#) [Async iteration and generators](#) [Asynchronous iteration](#) allow us to iterate over data that comes asynchronously, on-demand. Like, for instance, when we download something chunk-by-chunk over a network. And asynchronous generators make it even more convenient. Let's see a simple example first, to grasp the syntax, and then review a real-life use case.

Recall iterables Let's recall the topic about iterables.

The idea is that we have an object, such as `range` here:

```
let range = {
  from: 1,
  to: 5
};
```

...And we'd like to use `for..of` loop on it, such as `for(value of range)`, to get values from 1 to 5.

In other words, we want to add an iteration ability to the object.

That can be implemented using a special method with the name `Symbol.iterator`:

This method is called in by the `for..of` construct when the loop is started, and it should

return an object with the next method.

For each iteration, the next() method is invoked for the next value.

The next() should return a value in the form {done: true/false, value:<loop value>}, where done:true means the end of the loop.

Here's an implementation for the iterable range:

```
    let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() { // called once, in the beginning of for..of
    return {
      current: this.from,
      last: this.to,

      next() { // called every iteration, to get the next value
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for(let value of range) {
  alert(value); // 1 then 2, then 3, then 4, then 5
}
```

If anything is unclear, please visit the chapter Iterables, it gives all the details about regular iterables.

Async iterables Asynchronous iteration is needed when values come asynchronously: after `setTimeout` or another kind of delay.

The most common case is that the object needs to make a network request to deliver the next value, we'll see a real-life example of it a bit later.

To make an object iterable asynchronously:

Use `Symbol.asyncIterator` instead of `Symbol.iterator`.

The `next()` method should return a promise (to be fulfilled with the next value).

The `async` keyword handles it, we can simply make `async next()`.

To iterate over such an object, we should use a `for await (let item of iterable)` loop.

Note the `await` word.

As a starting example, let's make an iterable range object, similar like the one before, but now it will return values asynchronously, one per second.

All we need to do is to perform a few replacements in the code above:

```
let range = {
  from: 1,
  to: 5,

  [Symbol.asyncIterator]() { // (1)
    return {
      current: this.from,
      last: this.to,

      async next() { // (2)

        // note: we can use "await" inside the async next:
        await new Promise(resolve => setTimeout(resolve, 1000)); // (3)
      }
    }
  }
}
```

```

    if (this.current <= this.last) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};
}
};

```

```

(async () => {

  for await (let value of range) { // (4)
    alert(value); // 1,2,3,4,5
  }

})();

```

As we can see, the structure is similar to regular iterators:

To make an object asynchronously iterable, it must have a method `Symbol.asyncIterator` (1).

This method must return the object with `next()` method returning a promise (2).

The `next()` method doesn't have to be `async`, it may be a regular method returning a promise, but `async` allows us to use `await`, so that's convenient. Here we just delay for a second (3).

To iterate, we use `for await (let value of range)` (4), namely add "await" after "for". It calls `range[Symbol.asyncIterator]()` once, and then its `next()` for values.

Here's a small table with the differences:

Iterators
Async iterators

Object method to provide iterator
<code>Symbol.iterator</code>
<code>Symbol.asyncIterator</code>

next() return value is
any value
Promise

to loop, use
for..of
for await..of

The spread syntax ... doesn't work asynchronously
Features that require regular, synchronous iterators, don't work with
asynchronous ones.
For instance, a spread syntax won't work:

```
alert( [...range] ); // Error, no Symbol.iterator
```

That's natural, as it expects to find Symbol.iterator, not Symbol.asyncIterator.
It's also the case for for..of: the syntax without await needs Symbol.iterator.

Recall generators Now let's recall generators, as they allow to make iteration code much shorter. Most of the time, when we'd like to make an iterable, we'll use generators. For sheer simplicity, omitting some important stuff, they are "functions that generate (yield) values". They are explained in detail in the chapter Generators. Generators are labelled with function* (note the star) and use yield to generate a value, then we can use for..of to loop over them. This example generates a sequence of values from start to end:

```

    function* generateSequence(start, end) {
    for (let i = start; i <= end; i++) {
        yield i;
    }
}

for(let value of generateSequence(1, 5)) {
    alert(value); // 1, then 2, then 3, then 4, then 5
}

```

As we already know, to make an object iterable, we should add `Symbol.iterator` to it.

```

    let range = {
    from: 1,
    to: 5,
    [Symbol.iterator]() {
        return <object with next to make range iterable>
    }
}

```

A common practice for `Symbol.iterator` is to return a generator, it makes the code shorter, as you can see:

```

    let range = {
    from: 1,
    to: 5,

```

```

*[Symbol.iterator]() { // a shorthand for [Symbol.iterator]: function*()
  for(let value = this.from; value <= this.to; value++) {
    yield value;
  }
}
};

for(let value of range) {
  alert(value); // 1, then 2, then 3, then 4, then 5
}

```

Please see the chapter Generators if you'd like more details.

In regular generators we can't use `await`. All values must come synchronously, as required by the `for..of` construct.

What if we'd like to generate values asynchronously? From network requests, for instance.

Let's switch to asynchronous generators to make it possible.

Async generators (finally) For most practical applications, when we'd like to make an object that asynchronously generates a sequence of values, we can use an asynchronous generator.

The syntax is simple: prepend `function*` with `async`. That makes the generator asynchronous.

And then use `for await (...)` to iterate over it, like this:

```

async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {

    // Wow, can use await!
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }
}

```

```

    }
}

(async () => {

    let generator = generateSequence(1, 5);
    for await (let value of generator) {
        alert(value); // 1, then 2, then 3, then 4, then 5 (with delay between)
    }

})();

```

As the generator is asynchronous, we can use `await` inside it, rely on promises, perform network requests and so on.

Under-the-hood difference

Technically, if you're an advanced reader who remembers the details about generators, there's an internal difference. For async generators, the `generator.next()` method is asynchronous, it returns promises. In a regular generator we'd use `result = generator.next()` to get values. In an async generator, we should add `await`, like this:

```
result = await generator.next(); // result = {value: ..., done: true/false}
```

That's why async generators work with `for await...of`.

Async iterable rangeRegular generators can be used as `Symbol.iterator` to make the iteration code shorter.

Similar to that, async generators can be used as `Symbol.asyncIterator` to implement the asynchronous iteration.

For instance, we can make the range object generate values asynchronously, once per second, by replacing synchronous `Symbol.iterator` with asynchronous `Symbol.asyncIterator`:

```

    let range = {
from: 1,
to: 5,

// this line is same as [Symbol.asyncIterator]: async function*() {
async *[Symbol.asyncIterator]() {
  for(let value = this.from; value <= this.to; value++) {

    // make a pause between values, wait for something
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield value;
  }
}
};

(async () => {

  for await (let value of range) {
    alert(value); // 1, then 2, then 3, then 4, then 5
  }

})();

```

Now values come with a delay of 1 second between them.

Please note:

Technically, we can add both `Symbol.iterator` and `Symbol.asyncIterator` to the object, so it's both synchronously (`for..of`) and asynchronously (`for await..of`) iterable. In practice though, that would be a weird thing to do.

Real-life example: paginated data So far we've seen basic examples, to gain understanding. Now let's review a real-life use case.

There are many online services that deliver paginated data. For instance, when we need a list of users, a request returns a pre-defined count (e.g. 100 users) – “one page”, and provides a URL to the next page.

This pattern is very common. It's not about users, but just about anything.

For instance, GitHub allows us to retrieve commits in the same, paginated fashion:

We should make a request to fetch in the form `https://api.github.com/repos/<repo>/commits`.

It responds with a JSON of 30 commits, and also provides a link to the next page in the Link header.

Then we can use that link for the next request, to get more commits, and so on.

For our code, we'd like to have a simpler way to get commits.

Let's make a function `fetchCommits(repo)` that gets commits for us, making requests whenever needed. And let it care about all pagination stuff. For us it'll be a simple async iteration for `await..of`.

So the usage will be like this:

```
    for await (let commit of fetchCommits("username/repository")) {  
  // process commit  
}
```

Here's such function, implemented as async generator:

```
    async function* fetchCommits(repo) {  
  let url = `https://api.github.com/repos/${repo}/commits`;  
  
  while (url) {  
    const response = await fetch(url, { // (1)  
      headers: {'User-Agent': 'Our script'}, // github needs any user-agent header  
    });  
  
    const body = await response.json(); // (2) response is JSON (array of commits)  
  
    // (3) the URL of the next page is in the headers, extract it  
    let nextPage = response.headers.get('Link').match(/<(.*?)>; rel="next"/);  
    nextPage = nextPage?.[1];  
  
    url = nextPage;  
  
    for(let commit of body) { // (4) yield commits one by one, until the page ends  
      yield commit;  
    }  
  }  
}
```

```
}  
}  
}
```

More explanations about how it works:

We use the browser fetch method to download the commits.

The initial URL is `https://api.github.com/repos/<repo>/commits`, and the next page will be in the Link header of the response.

The fetch method allows us to supply authorization and other headers if needed – here GitHub requires User-Agent.

The commits are returned in JSON format.

We should get the next page URL from the Link header of the response. It has a special format, so we use a regular expression for that (we will learn this feature in Regular expressions).

The next page URL may look like `https://api.github.com/repositories/93253246/commits?page=2`. It's generated by GitHub itself.

Then we yield the received commits one by one, and when they finish, the next `while(url)` iteration will trigger, making one more request.

An example of use (shows commit authors in console):

```

    (async () => {
    let count = 0;

    for await (const commit of fetchCommits('javascript-tutorial/en.javascript.info')) {

        console.log(commit.author.login);

        if (++count == 100) { // let's stop at 100 commits
            break;
        }
    }

    })();

// Note: If you are running this in an external sandbox, you'll need to paste here the
function fetchCommits described above

```

That's just what we wanted.

The internal mechanics of paginated requests is invisible from the outside. For us it's just an async generator that returns commits.

SummaryRegular iterators and generators work fine with the data that doesn't take time to generate.

When we expect the data to come asynchronously, with delays, their async counterparts can be used, and for await..of instead of for..of.

Syntax differences between async and regular iterators:

Iterable
 Async Iterable

Method to provide iterator
 Symbol.iterator
 Symbol.asyncIterator

next() return value is
 {value:..., done: true/false}

Promise that resolves to {value:..., done: true/false}

Syntax differences between async and regular generators:

Generators

Async generators

Declaration

function*

async function*

next() return value is

{value:..., done: true/false}

Promise that resolves to {value:..., done: true/false}

In web-development we often meet streams of data, when it flows chunk-by-chunk. For instance, downloading or uploading a big file.

We can use async generators to process such data. It's also noteworthy that in some environments, like in browsers, there's also another API called Streams, that provides special interfaces to work with such streams, to transform the data and to pass it from one stream to another (e.g. download from one place and immediately send elsewhere).

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines –

wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};

Object.assign(this.page, {"url":"https://javascript.info/async-iterators-generators","identifier":"Vasync-iterators-generators"}); };var DisqusShortname =

"javascriptinfo";var DisqusEnabled = true;

TutorialThe JavaScript language{"@context":"https://

schema.org","@type":"BreadcrumbList","itemListElement":

[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},

{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://

javascript.info/js"}]}ModulesModules, introductionExport and ImportDynamic importsCtrl

+ !•Previous lessonCtrl + !'Next lessonShareTutorial map
TutorialThe JavaScript languageModules{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://
javascript.info/js"},{"@type":"ListItem","position":3,"name":"Modules","item":"https://
javascript.info/modules"}]}October 14, 2022Modules, introductionAs 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 – one of the most ancient module systems, initially implemented by the library
require.js.

CommonJS – the module system created for Node.js server.

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?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:

```
// Ø=ÜÁ sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

...Then another file may import and use it:

```
// Ø=ÜÁ 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:

```
Resultsay.jsindex.htmlexport function sayHi(user) {
  return `Hello, ${user}!`;
}<!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 or use the "live server" capability of your editor, such as VS Code Live Server Extension to test modules.

Core module featuresWhat's different in modules, compared to "regular" scripts?

There are core features, valid both for browser and server-side JavaScript.

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 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):

```

Resulthello.jsuser.jsindex.htmlalert(user); // no such variable (each module has
independent variables)let user = "John";<!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:

```

Resulthello.jsuser.jsindex.htmlimport {user} from './user.js';

```

```

document.body.innerHTML = user; // Johnexport let user = "John";<!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. 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:

```
// Ø=ÜÁ alert.js
alert("Module is evaluated!");
```

```
// Import the same module from different files
```

```
// Ø=ÜÁ 1.js
import `./alert.js`; // Module is evaluated!
```

```
// Ø=ÜÁ 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:

```
// Ø=ÜÁ 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:

```
// Ø=ÜÁ 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// Ø=ÜÁ 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:

A module exports some means of configuration, e.g. a configuration object.

On the first import we initialize it, write to its properties. The top-level application script may do that.

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:

```
// Ø=ÜÁ 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`:

```
// Ø=ÜÁ 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:

```
// Ø=ÜÁ another.js
import {sayHi} from './admin.js';
```

```
sayHi(); // Ready to serve, Pete!
```

`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 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 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 Module scripts are always deferred, same effect as defer attribute (described in the chapter Scripts: 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. 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 External scripts that have type="module" are different in two aspects:

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>
```

External scripts that are fetched from another origin (e.g. another site) require CORS headers, as described in the chapter Fetch: Cross-Origin Requests. 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 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” 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 In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as Webpack 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:

Take a “main” module, the one intended to be put in `<script type="module">` in HTML. Analyze its dependencies: imports and then imports of imports etc.

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.

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.

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 To summarize, the core concepts are:

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.

Modules have their own, local top-level scope and interchange functionality via import/export.

Modules always use strict.

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 to bundle modules together for performance and other reasons.

In the next chapter we’ll see more examples of modules, and how things can be exported/imported.

Ctrl + !•Previous lessonCtrl + !•Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https://www.javascript.info/modules-intro","identifier":"V

```
modules-intro")); };var Disqus_shortcode = "javascriptinfo";var Disqus_enabled = true;
TutorialThe JavaScript languageModules{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://
javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Modules","item":"https://
javascript.info/modules"}]]}October 1, 2022Export and ImportExport and import
directives have several syntax variants.
```

In the previous article we saw a simple use, now let's explore more examples.

Export before declarationsWe can label any declaration as exported by placing export before it, be it a variable, function or a class.

For instance, here all exports are valid:

```
// export an array
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

No semicolons after export class/function

Please note that export before a class or a function does not make it a function expression. It's still a function declaration, albeit exported.

Most JavaScript style guides don't recommend semicolons after function and class declarations.

That's why there's no need for a semicolon at the end of export class and export function:

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
} // no ; at the end
```

Export apart from declarations Also, we can put export separately.
Here we first declare, and then export:

```
// Ø=ÜÁ say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // a list of exported variables
```

...Or, technically we could put export above functions as well.
Import * Usually, we put a list of what to import in curly braces import {...}, like this:

```
// Ø=ÜÁ main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

But if there's a lot to import, we can import everything as an object using import * as <obj>, for instance:

```
// Ø=ÜÁ main.js
import * as say from './say.js';
```

```
say.sayHi('John');  
say.sayBye('John');
```

At first sight, “import everything” seems such a cool thing, short to write, why should we ever explicitly list what we need to import? Well, there are few reasons.

Explicitly listing what to import gives shorter names: `sayHi()` instead of `say.sayHi()`. Explicit list of imports gives better overview of the code structure: what is used and where. It makes code support and refactoring easier.

Don't be afraid to import too much

Modern build tools, such as webpack and others, bundle modules together and optimize them to speedup loading. They also removed unused imports. For instance, if you import `*` as library from a huge code library, and then use only few methods, then unused ones will not be included into the optimized bundle.

Import “as” We can also use `as` to import under different names.

For instance, let's import `sayHi` into the local variable `hi` for brevity, and import `sayBye` as `bye`:

```
// ð=ÜÁ main.js  
import {sayHi as hi, sayBye as bye} from './say.js';  
  
hi('John'); // Hello, John!  
bye('John'); // Bye, John!
```

Export “as” The similar syntax exists for export. Let's export functions as `hi` and `bye`:

```
// ð=ÜÁ say.js  
...  
export {sayHi as hi, sayBye as bye};
```


Now hi and bye are official names for outsiders, to be used in imports:

```
// Ø=ÜÁ main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

Export defaultIn practice, there are mainly two kinds of modules.

Modules that contain a library, pack of functions, like say.js above.

Modules that declare a single entity, e.g. a module user.js exports only class User.

Mostly, the second approach is preferred, so that every “thing” resides in its own module.

Naturally, that requires a lot of files, as everything wants its own module, but that's not a problem at all. Actually, code navigation becomes easier if files are well-named and structured into folders.

Modules provide a special export default (“the default export”) syntax to make the “one thing per module” way look better.

Put export default before the entity to export:

```
// Ø=ÜÁ user.js
export default class User { // just add "default"
  constructor(name) {
    this.name = name;
  }
}
```

There may be only one export default per file.

...And then import it without curly braces:

```
// ð=ÜÁ main.js
import User from './user.js'; // not {User}, just User

new User('John');
```

Imports without curly braces look nicer. A common mistake when starting to use modules is to forget curly braces at all. So, remember, import needs curly braces for named exports and doesn't need them for the default one.

Named export
Default export

```
export class User {...}
export default class User {...}
```

```
import {User} from ...
import User from ...
```

Technically, we may have both default and named exports in a single module, but in practice people usually don't mix them. A module has either named exports or the default one.

As there may be at most one default export per file, the exported entity may have no name.

For instance, these are all perfectly valid default exports:

```
export default class { // no class name
  constructor() { ... }
}
```

```
    export default function(user) { // no function name
    alert(`Hello, ${user}!`);
}
```

```
    // export a single value, without making a variable
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Not giving a name is fine, because there is only one export default per file, so import without curly braces knows what to import.
Without default, such an export would give an error:

```
    export class { // Error! (non-default export needs a name)
    constructor() {}
}
```

The “default” name
In some situations the default keyword is used to reference the default export.
For example, to export a function separately from its definition:

```
    function sayHi(user) {
    alert(`Hello, ${user}!`);
}
```

```
// same as if we added "export default" before the function
export {sayHi as default};
```

Or, another situation, let's say a module `user.js` exports one main "default" thing, and a few named ones (rarely the case, but it happens):

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Here's how to import the default export along with a named one:

```
// main.js
import {default as User, sayHi} from './user.js';

new User('John');
```

And, finally, if importing everything `*` as an object, then the default property is exactly the default export:

```
// main.js
import * as user from './user.js';

let User = user.default; // the default export
new User('John');
```

A word against default exports
Named exports are explicit. They exactly name what

they import, so we have that information from them; that's a good thing. Named exports force us to use exactly the right name to import:

```
import {User} from './user.js';  
// import {MyUser} won't work, the name must be {User}
```

...While for a default export, we always choose the name when importing:

```
import User from './user.js'; // works  
import MyUser from './user.js'; // works too  
// could be import Anything... and it'll still work
```

So team members may use different names to import the same thing, and that's not good.

Usually, to avoid that and keep the code consistent, there's a rule that imported variables should correspond to file names, e.g:

```
import User from './user.js';  
import LoginForm from './loginForm.js';  
import func from '/path/to/func.js';  
...
```

Still, some teams consider it a serious drawback of default exports. So they prefer to always use named exports. Even if only a single thing is exported, it's still exported under a name, without default.

That also makes re-export (see below) a little bit easier.

Re-export“Re-export” syntax `export ... from ...` allows to import things and immediately export them (possibly under another name), like this:

```
export {sayHi} from './say.js'; // re-export sayHi
```

```
export {default as User} from './user.js'; // re-export default
```

Why would that be needed? Let's see a practical use case.

Imagine, we're writing a "package": a folder with a lot of modules, with some of the functionality exported outside (tools like NPM allow us to publish and distribute such packages, but we don't have to use them), and many modules are just "helpers", for internal use in other package modules.

The file structure could be like this:

```
auth/  
index.js  
user.js  
helpers.js  
tests/  
  login.js  
providers/  
  github.js  
  facebook.js  
...
```

We'd like to expose the package functionality via a single entry point.

In other words, a person who would like to use our package, should import only from the "main file" `auth/index.js`.

Like this:

```
import {login, logout} from 'auth/index.js'
```

The "main file", `auth/index.js` exports all the functionality that we'd like to provide in our package.

The idea is that outsiders, other programmers who use our package, should not meddle with its internal structure, search for files inside our package folder. We export only

what's necessary in `auth/index.js` and keep the rest hidden from prying eyes. As the actual exported functionality is scattered among the package, we can import it into `auth/index.js` and export from it:

```
// auth/index.js

// import login/logout and immediately export them
import {login, logout} from './helpers.js';
export {login, logout};

// import default as User and export it
import User from './user.js';
export {User};
...
```

Now users of our package can import `{login}` from `"auth/index.js"`. The syntax `export ... from ...` is just a shorter notation for such import-export:

```
// auth/index.js

// re-export login/logout
export {login, logout} from './helpers.js';

// re-export the default export as User
export {default as User} from './user.js';
...
```

The notable difference of `export ... from` compared to `import/export` is that re-exported modules aren't available in the current file. So inside the above example of `auth/index.js` we can't use re-exported `login/logout` functions.

Re-exporting the default exportThe default export needs separate handling when re-exporting.

Let's say we have `user.js` with the export default class `User` and would like to re-export it:

```
// Ø=ÜÁ user.js
export default class User {
  // ...
}
```

We can come across two problems with it:

export User from './user.js' won't work. That would lead to a syntax error.
To re-export the default export, we have to write export {default as User}, as in the example above.

export * from './user.js' re-exports only named exports, but ignores the default one.
If we'd like to re-export both named and default exports, then two statements are needed:

```
export * from './user.js'; // to re-export named exports
export {default} from './user.js'; // to re-export the default export
```

Such oddities of re-exporting a default export are one of the reasons why some developers don't like default exports and prefer named ones.
SummaryHere are all types of export that we covered in this and previous articles.
You can check yourself by reading them and recalling what they mean:

Before declaration of a class/function/...:

```
export [default] class/function/variable ...
```

Standalone export:

```
export {x [as y], ...}.
```

Re-export:


```
export {x [as y], ...} from "module"  
export * from "module" (doesn't re-export default).  
export {default [as y]} from "module" (re-export default).
```

Import:

Importing named exports:

```
import {x [as y], ...} from "module"
```

Importing the default export:

```
import x from "module"  
import {default as x} from "module"
```

Import all:

```
import * as obj from "module"
```

Import the module (its code runs), but do not assign any of its exports to variables:

```
import "module"
```

We can put import/export statements at the top or at the bottom of a script, that doesn't matter.

So, technically this code is fine:

```
    sayHi();  
  
// ...  
  
import {sayHi} from './say.js'; // import at the end of the file
```

In practice imports are usually at the start of the file, but that's only for more convenience.

Please note that import/export statements don't work if inside {...}.

A conditional import, like this, won't work:

```
if (something) {  
  import {sayHi} from "./say.js"; // Error: import must be at top level  
}
```

...But what if we really need to import something conditionally? Or at the right time? Like, load a module upon request, when it's really needed?

We'll see dynamic imports in the next article.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/import-export", "identifier": "import-export" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageModules{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement": [{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"}, {"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"}, {"@type": "ListItem", "position": 3, "name": "Modules", "item": "https://javascript.info/modules"}]}February 8, 2020Dynamic importsExport and import statements that we covered in previous chapters are called “static”. The syntax is very simple and strict.

First, we can't dynamically generate any parameters of import.

The module path must be a primitive string, can't be a function call. This won't work:

```
import ... from getModuleName(); // Error, only from "string" is allowed
```

Second, we can't import conditionally or at run-time:

```

    if(...) {
import ...; // Error, not allowed!
}

{
import ...; // Error, we can't put import in any block
}

```

That's because import/export aim to provide a backbone for the code structure. That's a good thing, as code structure can be analyzed, modules can be gathered and bundled into one file by special tools, unused exports can be removed ("tree-shaken"). That's possible only because the structure of imports/exports is simple and fixed. But how can we import a module dynamically, on-demand? The import() expressionThe import(module) expression loads the module and returns a promise that resolves into a module object that contains all its exports. It can be called from any place in the code. We can use it dynamically in any place of the code, for instance:

```

    let modulePath = prompt("Which module to load?");

import(modulePath)
  .then(obj => <module object>)
  .catch(err => <loading error, e.g. if no such module>)

```

Or, we could use let module = await import(modulePath) if inside an async function. For instance, if we have the following module say.js:

```

// Ø=ÜÁ say.js
export function hi() {
  alert(`Hello`);
}

export function bye() {
  alert(`Bye`);
}

```

```
}
```

...Then dynamic import can be like this:

```
let {hi, bye} = await import('./say.js');  
  
hi();  
bye();
```

Or, if say.js has the default export:

```
// Ø=ÜÁ say.js  
export default function() {  
  alert("Module loaded (export default)!");  
}
```

...Then, in order to access it, we can use default property of the module object:

```
let obj = await import('./say.js');  
let say = obj.default;  
// or, in one line: let {default: say} = await import('./say.js');  
  
say();
```

Here's the full example:

```
Resultsay.jsindex.htmlexport function hi() {  
  alert(`Hello`);  
}
```

```

export function bye() {
  alert(` Bye`);
}

export default function() {
  alert("Module loaded (export default)!");
}<!doctype html>
<script>
  async function load() {
    let say = await import('./say.js');
    say.hi(); // Hello!
    say.bye(); // Bye!
    say.default(); // Module loaded (export default)!
  }
</script>
<button onclick="load()">Click me</button>
  Please note:
  Dynamic imports work in regular scripts, they don't require script type="module".

```

Please note:
 Although `import()` looks like a function call, it's a special syntax that just happens to use parentheses (similar to `super()`).
 So we can't copy `import` to a variable or use `call/apply` with it. It's not a function.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
 Object.assign(this.page, {"url":"https://javascript.info/modules-dynamic-imports","identifier":"modules-dynamic-imports"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
 TutorialThe JavaScript language{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
 [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}]}MiscellaneousProxy and ReflectEval: run a code
 stringCurryingReference TypeBigIntUnicode, String internalsCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map
 TutorialThe JavaScript languageMiscellaneous{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
 [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}],

```
{"@type":"ListItem","position":3,"name":"Miscellaneous","item":"https://javascript.info/js-misc"}}}
```

June 26, 2022

Proxy and Reflect

A Proxy object wraps another object and intercepts operations, like reading/writing properties and others, optionally handling them on its own, or transparently allowing the object to handle them.

Proxies are used in many libraries and some browser frameworks. We'll see many practical applications in this article.

Proxy

The syntax:

```
let proxy = new Proxy(target, handler)
```

target – is an object to wrap, can be anything, including functions.

handler – proxy configuration: an object with “traps”, methods that intercept operations.

– e.g. get trap for reading a property of target, set trap for writing a property into target, and so on.

For operations on proxy, if there's a corresponding trap in handler, then it runs, and the proxy has a chance to handle it, otherwise the operation is performed on target.

As a starting example, let's create a proxy without any traps:

```
let target = {};  
let proxy = new Proxy(target, {}); // empty handler  
  
proxy.test = 5; // writing to proxy (1)  
alert(target.test); // 5, the property appeared in target!  
  
alert(proxy.test); // 5, we can read it from proxy too (2)  
  
for(let key in proxy) alert(key); // test, iteration works (3)
```

As there are no traps, all operations on proxy are forwarded to target.

A writing operation `proxy.test=` sets the value on target.

A reading operation `proxy.test` returns the value from target.

Iteration over proxy returns values from target.

As we can see, without any traps, proxy is a transparent wrapper around target.

Proxy is a special “exotic object”. It doesn’t have own properties. With an empty handler it transparently forwards operations to target.

To activate more capabilities, let’s add traps.

What can we intercept with them?

For most operations on objects, there’s a so-called “internal method” in the JavaScript specification that describes how it works at the lowest level. For instance `[[Get]]`, the internal method to read a property, `[[Set]]`, the internal method to write a property, and so on. These methods are only used in the specification, we can’t call them directly by name.

Proxy traps intercept invocations of these methods. They are listed in the Proxy specification and in the table below.

For every internal method, there’s a trap in this table: the name of the method that we can add to the handler parameter of new Proxy to intercept the operation:

Internal Method
Handler Method
Triggers when...

<code>[[Get]]</code>
<code>get</code>
reading a property

<code>[[Set]]</code>
<code>set</code>
writing to a property

[[HasProperty]]

has

in operator

[[Delete]]

deleteProperty

delete operator

[[Call]]

apply

function call

[[Construct]]

construct

new operator

[[GetPrototypeOf]]

getPrototypeOf

Object.getPrototypeOf

[[SetPrototypeOf]]

setPrototypeOf

Object.setPrototypeOf

[[IsExtensible]]

isExtensible

Object.isExtensible

[[PreventExtensions]]

preventExtensions

Object.preventExtensions

[[DefineOwnProperty]]

defineProperty

Object.defineProperty, Object.defineProperties

[[GetOwnProperty]]
getOwnPropertyDescriptor
Object.getOwnPropertyDescriptor, for..in, Object.keys/values/entries

[[OwnPropertyKeys]]
ownKeys
Object.getOwnPropertyNames, Object.getOwnPropertySymbols, for..in, Object.keys/
values/entries

Invariants

JavaScript enforces some invariants – conditions that must be fulfilled by internal methods and traps.

Most of them are for return values:

[[Set]] must return true if the value was written successfully, otherwise false.
[[Delete]] must return true if the value was deleted successfully, otherwise false.
...and so on, we'll see more in examples below.

There are some other invariants, like:

[[GetPrototypeOf]], applied to the proxy object must return the same value as
[[GetPrototypeOf]] applied to the proxy object's target object. In other words, reading
prototype of a proxy must always return the prototype of the target object.

Traps can intercept these operations, but they must follow these rules.
Invariants ensure correct and consistent behavior of language features. The full
invariants list is in the specification. You probably won't violate them if you're not doing
something weird.

Let's see how that works in practical examples.

Default value with "get" trap
The most common traps are for reading/writing properties.
To intercept reading, the handler should have a method get(target, property, receiver).
It triggers when a property is read, with following arguments:

target – is the target object, the one passed as the first argument to new Proxy,
property – property name,
receiver – if the target property is a getter, then receiver is the object that's going to be
used as this in its call. Usually that's the proxy object itself (or an object that inherits
from it, if we inherit from proxy). Right now we don't need this argument, so it will be
explained in more detail later.

Let's use get to implement default values for an object.

We'll make a numeric array that returns 0 for nonexistent values. Usually when one tries to get a non-existing array item, they get undefined, but we'll wrap a regular array into the proxy that traps reading and returns 0 if there's no such property:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // default value
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (no such item)
```

As we can see, it's quite easy to do with a get trap. We can use Proxy to implement any logic for "default" values. Imagine we have a dictionary, with phrases and their translations:

```

    let dictionary = {
      'Hello': 'Hola',
      'Bye': 'Adiós'
    };

    alert( dictionary['Hello'] ); // Hola
    alert( dictionary['Welcome'] ); // undefined

```

Right now, if there's no phrase, reading from dictionary returns undefined. But in practice, leaving a phrase untranslated is usually better than undefined. So let's make it return an untranslated phrase in that case instead of undefined. To achieve that, we'll wrap dictionary in a proxy that intercepts reading operations:

```

    let dictionary = {
      'Hello': 'Hola',
      'Bye': 'Adiós'
    };

    dictionary = new Proxy(dictionary, {
      get(target, phrase) { // intercept reading a property from dictionary
        if (phrase in target) { // if we have it in the dictionary
          return target[phrase]; // return the translation
        } else {
          // otherwise, return the non-translated phrase
          return phrase;
        }
      }
    });

    // Look up arbitrary phrases in the dictionary!
    // At worst, they're not translated.

```

```
alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome to Proxy'] ); // Welcome to Proxy (no translation)
```

Please note:

Please note how the proxy overwrites the variable:

```
dictionary = new Proxy(dictionary, ...);
```

The proxy should totally replace the target object everywhere. No one should ever reference the target object after it got proxied. Otherwise it's easy to mess up.

Validation with “set” trap Let's say we want an array exclusively for numbers. If a value of another type is added, there should be an error.

The set trap triggers when a property is written.

set(target, property, value, receiver):

target – is the target object, the one passed as the first argument to new Proxy,

property – property name,

value – property value,

receiver – similar to get trap, matters only for setter properties.

The set trap should return true if setting is successful, and false otherwise (triggers TypeError).

Let's use it to validate new values:

```
let numbers = [];
```

```

numbers = new Proxy(numbers, { // (*)
  set(target, prop, val) { // to intercept property writing
    if (typeof val == 'number') {
      target[prop] = val;
      return true;
    } else {
      return false;
    }
  }
});

numbers.push(1); // added successfully
numbers.push(2); // added successfully
alert("Length is: " + numbers.length); // 2

numbers.push("test"); // TypeError ('set' on proxy returned false)

alert("This line is never reached (error in the line above)");

```

Please note: the built-in functionality of arrays is still working! Values are added by push. The length property auto-increases when values are added. Our proxy doesn't break anything.

We don't have to override value-adding array methods like push and unshift, and so on, to add checks in there, because internally they use the `[[Set]]` operation that's intercepted by the proxy.

So the code is clean and concise.

Don't forget to return true

As said above, there are invariants to be held.

For set, it must return true for a successful write.

If we forget to do it or return any falsy value, the operation triggers `TypeError`.

Iteration with `"ownKeys"` and `"getOwnPropertyDescriptor"` `Object.keys`, `for..in` loop and most other methods that iterate over object properties use `[[OwnPropertyKeys]]` internal method (intercepted by `ownKeys` trap) to get a list of properties.

Such methods differ in details:

`Object.getPrototypeOfNames(obj)` returns non-symbol keys.

`Object.getPrototypeOfSymbols(obj)` returns symbol keys.

`Object.keys/values()` returns non-symbol keys/values with enumerable flag (property flags were explained in the article `Property flags and descriptors`).

`for..in` loops over non-symbol keys with enumerable flag, and also prototype keys.

...But all of them start with that list.

In the example below we use ownKeys trap to make for..in loop over user, and also Object.keys and Object.values, to skip properties starting with an underscore _:

```
    let user = {
  name: "John",
  age: 30,
  _password: "****"
};

user = new Proxy(user, {
  ownKeys(target) {
    return Object.keys(target).filter(key => !key.startsWith('_'));
  }
});

// "ownKeys" filters out _password
for(let key in user) alert(key); // name, then: age

// same effect on these methods:
alert( Object.keys(user) ); // name,age
alert( Object.values(user) ); // John,30
```

So far, it works.

Although, if we return a key that doesn't exist in the object, Object.keys won't list it:

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) {
    return ['a', 'b', 'c'];
  }
});

alert( Object.keys(user) ); // <empty>

```

Why? The reason is simple: `Object.keys` returns only properties with the `enumerable` flag. To check for it, it calls the internal method `[[GetOwnProperty]]` for every property to get its descriptor. And here, as there's no property, its descriptor is empty, no `enumerable` flag, so it's skipped.

For `Object.keys` to return a property, we need it to either exist in the object, with the `enumerable` flag, or we can intercept calls to `[[GetOwnProperty]]` (the trap `getOwnPropertyDescriptor` does it), and return a descriptor with `enumerable: true`. Here's an example of that:

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) { // called once to get a list of properties
    return ['a', 'b', 'c'];
  },

  getOwnPropertyDescriptor(target, prop) { // called for every property
    return {
      enumerable: true,
      configurable: true
    };
  }
});

```

```

    /* ...other flags, probable "value:..." */
  };
}

});

alert( Object.keys(user) ); // a, b, c

```

Let's note once again: we only need to intercept `[[GetOwnProperty]]` if the property is absent in the object.

Protected properties with “`deleteProperty`” and other traps

There's a widespread convention that properties and methods prefixed by an underscore `_` are internal. They shouldn't be accessed from outside the object. Technically that's possible though:

```

let user = {
  name: "John",
  _password: "secret"
};

alert(user._password); // secret

```

Let's use proxies to prevent any access to properties starting with `_`. We'll need the traps:

- `get` to throw an error when reading such property,
- `set` to throw an error when writing,
- `deleteProperty` to throw an error when deleting,
- `ownKeys` to exclude properties starting with `_` from `for..in` and methods like `Object.keys`.

Here's the code:


```

    let user = {
    name: "John",
    _password: "****"
  };

  user = new Proxy(user, {
    get(target, prop) {
      if (prop.startsWith('_')) {
        throw new Error("Access denied");
      }
      let value = target[prop];
      return (typeof value === 'function') ? value.bind(target) : value; // (*)
    },
    set(target, prop, val) { // to intercept property writing
      if (prop.startsWith('_')) {
        throw new Error("Access denied");
      } else {
        target[prop] = val;
        return true;
      }
    },
    deleteProperty(target, prop) { // to intercept property deletion
      if (prop.startsWith('_')) {
        throw new Error("Access denied");
      } else {
        delete target[prop];
        return true;
      }
    },
    ownKeys(target) { // to intercept property list
      return Object.keys(target).filter(key => !key.startsWith('_'));
    }
  });

```

```

// "get" doesn't allow to read _password
try {
  alert(user._password); // Error: Access denied
} catch(e) { alert(e.message); }

// "set" doesn't allow to write _password
try {
  user._password = "test"; // Error: Access denied
} catch(e) { alert(e.message); }

// "deleteProperty" doesn't allow to delete _password
try {
  delete user._password; // Error: Access denied
} catch(e) { alert(e.message); }

// "ownKeys" filters out _password
for(let key in user) alert(key); // name

```

Please note the important detail in the get trap, in the line (*):

```

    get(target, prop) {
// ...
let value = target[prop];
return (typeof value === 'function') ? value.bind(target) : value; // (*)
}

```

Why do we need a function to call `value.bind(target)`?

The reason is that object methods, such as `user.checkPassword()`, must be able to access `_password`:

```

    user = {
// ...
checkPassword(value) {
  // object method must be able to read _password
  return value === this._password;
}

```

```
}
```

A call to `user.checkPassword()` gets proxied `user` as `this` (the object before dot becomes `this`), so when it tries to access `this._password`, the get trap activates (it triggers on any property read) and throws an error.

So we bind the context of object methods to the original object, `target`, in the line (*). Then their future calls will use `target` as `this`, without any traps.

That solution usually works, but isn't ideal, as a method may pass the unproxied object somewhere else, and then we'll get messed up: where's the original object, and where's the proxied one?

Besides, an object may be proxied multiple times (multiple proxies may add different "tweaks" to the object), and if we pass an unwrapped object to a method, there may be unexpected consequences.

So, such a proxy shouldn't be used everywhere.

Private properties of a class

Modern JavaScript engines natively support private properties in classes, prefixed with `#`. They are described in the article [Private and protected properties and methods](#). No proxies required.

Such properties have their own issues though. In particular, they are not inherited.

"In range" with "has" trap Let's see more examples.

We have a range object:

```
let range = {  
  start: 1,  
  end: 10  
};
```

We'd like to use the `in` operator to check that a number is in range.

The `has` trap intercepts `in` calls.

`has(target, property)`

`target` – is the target object, passed as the first argument to `new Proxy`,

`property` – property name

Here's the demo:

```
    let range = {
      start: 1,
      end: 10
    };

    range = new Proxy(range, {
      has(target, prop) {
        return prop >= target.start && prop <= target.end;
      }
    });

    alert(5 in range); // true
    alert(50 in range); // false
```

Nice syntactic sugar, isn't it? And very simple to implement.
Wrapping functions: "apply" We can wrap a proxy around a function as well.
The `apply(target, thisArg, args)` trap handles calling a proxy as function:

`target` is the target object (function is an object in JavaScript),
`thisArg` is the value of `this`.
`args` is a list of arguments.

For example, let's recall `delay(f, ms)` decorator, that we did in the article [Decorators and forwarding, call/apply](#).

In that article we did it without proxies. A call to `delay(f, ms)` returned a function that forwards all calls to `f` after `ms` milliseconds.

Here's the previous, function-based implementation:

```

    function delay(f, ms) {
    // return a wrapper that passes the call to f after the timeout
    return function() { // (*)
        setTimeout(() => f.apply(this, arguments), ms);
    };
}

function sayHi(user) {
    alert(`Hello, ${user}!`);
}

// after this wrapping, calls to sayHi will be delayed for 3 seconds
sayHi = delay(sayHi, 3000);

sayHi("John"); // Hello, John! (after 3 seconds)

```

As we've seen already, that mostly works. The wrapper function (*) performs the call after the timeout. But a wrapper function does not forward property read/write operations or anything else. After the wrapping, the access is lost to properties of the original functions, such as name, length and others:

```

    function delay(f, ms) {
    return function() {
        setTimeout(() => f.apply(this, arguments), ms);
    };
}

```

```
function sayHi(user) {  
  alert(`Hello, ${user}!`);  
}
```

```
alert(sayHi.length); // 1 (function length is the arguments count in its declaration)
```

```
sayHi = delay(sayHi, 3000);
```

```
alert(sayHi.length); // 0 (in the wrapper declaration, there are zero arguments)
```

Proxy is much more powerful, as it forwards everything to the target object.
Let's use Proxy instead of a wrapping function:

```
function delay(f, ms) {  
  return new Proxy(f, {  
    apply(target, thisArg, args) {  
      setTimeout(() => target.apply(thisArg, args), ms);  
    }  
  });  
}
```

```
function sayHi(user) {  
  alert(`Hello, ${user}!`);  
}
```

```
sayHi = delay(sayHi, 3000);
```

```
alert(sayHi.length); // 1 (*) proxy forwards "get length" operation to the target
```

```
sayHi("John"); // Hello, John! (after 3 seconds)
```

The result is the same, but now not only calls, but all operations on the proxy are forwarded to the original function. So `sayHi.length` is returned correctly after the wrapping in the line (*).

We've got a "richer" wrapper.

Other traps exist: the full list is in the beginning of this article. Their usage pattern is similar to the above.

`Reflect` is a built-in object that simplifies creation of `Proxy`.

It was said previously that internal methods, such as `[[Get]]`, `[[Set]]` and others are specification-only, they can't be called directly.

The `Reflect` object makes that somewhat possible. Its methods are minimal wrappers around the internal methods.

Here are examples of operations and `Reflect` calls that do the same:

Operation	Reflect call	Internal method
-----------	--------------	-----------------

<code>obj[prop]</code>	<code>Reflect.get(obj, prop)</code>	<code>[[Get]]</code>
------------------------	-------------------------------------	----------------------

<code>obj[prop] = value</code>	<code>Reflect.set(obj, prop, value)</code>	<code>[[Set]]</code>
--------------------------------	--	----------------------

<code>delete obj[prop]</code>	<code>Reflect.deleteProperty(obj, prop)</code>	<code>[[Delete]]</code>
-------------------------------	--	-------------------------

<code>new F(value)</code>	<code>Reflect.construct(F, value)</code>	<code>[[Construct]]</code>
---------------------------	--	----------------------------

...		
...		
...		

For example:

```
let user = {};  
  
Reflect.set(user, 'name', 'John');  
  
alert(user.name); // John
```

In particular, Reflect allows us to call operators (new, delete...) as functions (Reflect.construct, Reflect.deleteProperty, ...). That's an interesting capability, but here another thing is important.

For every internal method, trappable by Proxy, there's a corresponding method in Reflect, with the same name and arguments as the Proxy trap.

So we can use Reflect to forward an operation to the original object.

In this example, both traps get and set transparently (as if they didn't exist) forward reading/writing operations to the object, showing a message:

```
let user = {  
  name: "John",  
};
```



```

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);
    return Reflect.get(target, prop, receiver); // (1)
  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);
    return Reflect.set(target, prop, val, receiver); // (2)
  }
});

```

```

let name = user.name; // shows "GET name"
user.name = "Pete"; // shows "SET name=Pete"

```

Here:

Reflect.get reads an object property.

Reflect.set writes an object property and returns true if successful, false otherwise.

That is, everything's simple: if a trap wants to forward the call to the object, it's enough to call Reflect.<method> with the same arguments.

In most cases we can do the same without Reflect, for instance, reading a property Reflect.get(target, prop, receiver) can be replaced by target[prop]. There are important nuances though.

Proxying a getterLet's see an example that demonstrates why Reflect.get is better. And we'll also see why get/set have the third argument receiver, that we didn't use before. We have an object user with _name property and a getter for it.

Here's a proxy around it:

```

let user = {
  _name: "Guest",
  get name() {

```

```

    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
});

alert(userProxy.name); // Guest

```

The get trap is “transparent” here, it returns the original property, and doesn’t do anything else. That’s enough for our example. Everything seems to be all right. But let’s make the example a little bit more complex. After inheriting another object admin from user, we can observe the incorrect behavior:

```

    let user = {
      _name: "Guest",
      get name() {
        return this._name;
      }
    };

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
}

```

```
};
```

```
// Expected: Admin  
alert(admin.name); // outputs: Guest (?!?)
```

Reading `admin.name` should return "Admin", not "Guest"! What's the matter? Maybe we did something wrong with the inheritance? But if we remove the proxy, then everything will work as expected. The problem is actually in the proxy, in the line (*).

When we read `admin.name`, as `admin` object doesn't have such own property, the search goes to its prototype.

The prototype is `userProxy`.

When reading `name` property from the proxy, its `get` trap triggers and returns it from the original object as `target[prop]` in the line (*).

A call to `target[prop]`, when `prop` is a getter, runs its code in the context `this=target`. So the result is `this._name` from the original object `target`, that is: from `user`.

To fix such situations, we need `receiver`, the third argument of `get` trap. It keeps the correct `this` to be passed to a getter. In our case that's `admin`.

How to pass the context for a getter? For a regular function we could use `call`/`apply`, but that's a getter, it's not "called", just accessed.

`Reflect.get` can do that. Everything will work right if we use it. Here's the corrected variant:

```
let user = {  
  _name: "Guest",
```

```

    get name() {
      return this._name;
    }
  };

  let userProxy = new Proxy(user, {
    get(target, prop, receiver) { // receiver = admin
      return Reflect.get(target, prop, receiver); // (*)
    }
  });

  let admin = {
    __proto__: userProxy,
    _name: "Admin"
  };

  alert(admin.name); // Admin

```

Now receiver that keeps a reference to the correct this (that is admin), is passed to the getter using Reflect.get in the line (*).
We can rewrite the trap even shorter:

```

    get(target, prop, receiver) {
      return Reflect.get(...arguments);
    }

```

Reflect calls are named exactly the same way as traps and accept the same arguments. They were specifically designed this way.

So, return Reflect... provides a safe no-brainer to forward the operation and make sure we don't forget anything related to that.

Proxy limitations Proxies provide a unique way to alter or tweak the behavior of the existing objects at the lowest level. Still, it's not perfect. There are limitations.

Built-in objects: Internal slots Many built-in objects, for example Map, Set, Date, Promise and others make use of so-called "internal slots".

These are like properties, but reserved for internal, specification-only purposes. For instance, Map stores items in the internal slot `[[MapData]]`. Built-in methods access them directly, not via `[[Get]]`/`[[Set]]` internal methods. So Proxy can't intercept that.

Why care? They're internal anyway!

Well, here's the issue. After a built-in object like that gets proxied, the proxy doesn't have these internal slots, so built-in methods will fail.
For example:

```
let map = new Map();  
  
let proxy = new Proxy(map, {});  
  
proxy.set('test', 1); // Error
```

Internally, a Map stores all data in its `[[MapData]]` internal slot. The proxy doesn't have such a slot. The built-in method `Map.prototype.set` method tries to access the internal property `this[[MapData]]`, but because `this=proxy`, can't find it in proxy and just fails.
Fortunately, there's a way to fix it:

```
let map = new Map();  
  
let proxy = new Proxy(map, {  
  get(target, prop, receiver) {  
    let value = Reflect.get(...arguments);  
    return typeof value == 'function' ? value.bind(target) : value;  
  }  
});
```

```
}  
});
```

```
proxy.set('test', 1);  
alert(proxy.get('test')); // 1 (works!)
```

Now it works fine, because `get` trap binds function properties, such as `map.set`, to the target object (`map`) itself.

Unlike the previous example, the value of this inside `proxy.set(...)` will be not `proxy`, but the original `map`. So when the internal implementation of `set` tries to access this. `[[MapData]]` internal slot, it succeeds.

Array has no internal slots

A notable exception: built-in `Array` doesn't use internal slots. That's for historical reasons, as it appeared so long ago.

So there's no such problem when proxying an array.

Private fields A similar thing happens with private class fields.

For example, `getName()` method accesses the private `#name` property and breaks after proxying:

```
class User {  
  #name = "Guest";  
  
  getName() {  
    return this.#name;  
  }  
}
```

```
let user = new User();
```

```
user = new Proxy(user, {});
```

```
alert(user.getName()); // Error
```

The reason is that private fields are implemented using internal slots. JavaScript does not use `[[Get]]`/`[[Set]]` when accessing them. In the call `getName()` the value of `this` is the proxied user, and it doesn't have the slot with private fields. Once again, the solution with binding the method makes it work:

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) : value;
  }
});

alert(user.getName()); // Guest
```

That said, the solution has drawbacks, as explained previously: it exposes the original object to the method, potentially allowing it to be passed further and breaking other proxied functionality. `Proxy != target` The proxy and the original object are different objects. That's natural, right?

So if we use the original object as a key, and then proxy it, then the proxy can't be found:

```
let allUsers = new Set();

class User {
  constructor(name) {
    this.name = name;
    allUsers.add(this);
  }
}

let user = new User("John");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false
```

As we can see, after proxying we can't find user in the set allUsers, because the proxy is a different object.

Proxies can't intercept a strict equality test ===

Proxies can intercept many operators, such as new (with construct), in (with has), delete (with deleteProperty) and so on.

But there's no way to intercept a strict equality test for objects. An object is strictly equal to itself only, and no other value.

So all operations and built-in classes that compare objects for equality will differentiate between the object and the proxy. No transparent replacement here.

Revocable proxies A revocable proxy is a proxy that can be disabled.

Let's say we have a resource, and would like to close access to it any moment.

What we can do is to wrap it into a revocable proxy, without any traps. Such a proxy will forward operations to object, and we can disable it at any moment. The syntax is:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

The call returns an object with the proxy and revoke function to disable it. Here's an example:

```
let object = {  
  data: "Valuable data"  
};  
  
let {proxy, revoke} = Proxy.revocable(object, {});  
  
// pass the proxy somewhere instead of object...  
alert(proxy.data); // Valuable data  
  
// later in our code  
revoke();  
  
// the proxy isn't working any more (revoked)  
alert(proxy.data); // Error
```

A call to `revoke()` removes all internal references to the target object from the proxy, so they are no longer connected. Initially, `revoke` is separate from `proxy`, so that we can pass `proxy` around while leaving `revoke` in the current scope.

We can also bind revoke method to proxy by setting `proxy.revoke = revoke`. Another option is to create a `WeakMap` that has proxy as the key and the corresponding revoke as the value, that allows to easily find revoke for a proxy:

```
let revokes = new WeakMap();

let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ..somewhere else in our code..
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Error (revoked)
```

We use `WeakMap` instead of `Map` here because it won't block garbage collection. If a proxy object becomes "unreachable" (e.g. no variable references it any more), `WeakMap` allows it to be wiped from memory together with its revoke that we won't need any more.

References

Specification: [Proxy](#).

MDN: [Proxy](#).

`SummaryProxy` is a wrapper around an object, that forwards operations on it to the object, optionally trapping some of them.

It can wrap any kind of object, including classes and functions.

The syntax is:

```
    let proxy = new Proxy(target, {  
/* traps */  
});
```

...Then we should use proxy everywhere instead of target. A proxy doesn't have its own properties or methods. It traps an operation if the trap is provided, otherwise forwards it to target object.

We can trap:

Reading (get), writing (set), deleting (deleteProperty) a property (even a non-existing one).

Calling a function (apply trap).

The new operator (construct trap).

Many other operations (the full list is at the beginning of the article and in the docs).

That allows us to create "virtual" properties and methods, implement default values, observable objects, function decorators and so much more.

We can also wrap an object multiple times in different proxies, decorating it with various aspects of functionality.

The Reflect API is designed to complement Proxy. For any Proxy trap, there's a Reflect call with same arguments. We should use those to forward calls to target objects.

Proxies have some limitations:

Built-in objects have "internal slots", access to those can't be proxied. See the workaround above.

The same holds true for private class fields, as they are internally implemented using slots. So proxied method calls must have the target object as this to access them.

Object equality tests `===` can't be intercepted.

Performance: benchmarks depend on an engine, but generally accessing a property using a simplest proxy takes a few times longer. In practice that only matters for some "bottleneck" objects though.

TasksError on reading non-existent propertyUsually, an attempt to read a non-existent property returns undefined.

Create a proxy that throws an error for an attempt to read of a non-existent property instead.

That can help to detect programming mistakes early.

Write a function `wrap(target)` that takes an object target and return a proxy that adds this functionality aspect.

That's how it should work:

```
    let user = {
      name: "John"
    };

    function wrap(target) {
      return new Proxy(target, {
        /* your code */
      });
    }

    user = wrap(user);

    alert(user.name); // John
    alert(user.age); // ReferenceError: Property doesn't exist: "age"
```

solution

```
    let user = {
      name: "John"
    };

    function wrap(target) {
      return new Proxy(target, {
        get(target, prop, receiver) {
          if (prop in target) {
            return Reflect.get(target, prop, receiver);
          } else {
            throw new ReferenceError(`Property doesn't exist: "${prop}"`);
          }
        }
      });
    }
```

```
}  
  
user = wrap(user);  
  
alert(user.name); // John  
alert(user.age); // ReferenceError: Property doesn't exist: "age"
```

Accessing array[-1] In some programming languages, we can access array elements using negative indexes, counted from the end.
Like this:

```
let array = [1, 2, 3];  
  
array[-1]; // 3, the last element  
array[-2]; // 2, one step from the end  
array[-3]; // 1, two steps from the end
```

In other words, array[-N] is the same as array[array.length - N].
Create a proxy to implement that behavior.
That's how it should work:

```
let array = [1, 2, 3];  
  
array = new Proxy(array, {  
  /* your code */  
});  
  
alert( array[-1] ); // 3  
alert( array[-2] ); // 2  
  
// Other array functionality should be kept "as is"
```

solution

```

    let array = [1, 2, 3];

array = new Proxy(array, {
  get(target, prop, receiver) {
    if (prop < 0) {
      // even if we access it like arr[1]
      // prop is a string, so need to convert it to number
      prop = +prop + target.length;
    }
    return Reflect.get(target, prop, receiver);
  }
});

```

```

alert(array[-1]); // 3
alert(array[-2]); // 2

```

Observable Create a function `makeObservable(target)` that “makes the object observable” by returning a proxy.
Here’s how it should work:

```

function makeObservable(target) {
/* your code */

```

```

}

let user = {};
user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John"; // alerts: SET name=John

```

In other words, an object returned by `makeObservable` is just like the original one, but also has the method `observe(handler)` that sets handler function to be called on any property change.

Whenever a property changes, `handler(key, value)` is called with the name and value of the property.

P.S. In this task, please only take care about writing to a property. Other operations can be implemented in a similar way.

The solution consists of two parts:

Whenever `.observe(handler)` is called, we need to remember the handler somewhere, to be able to call it later. We can store handlers right in the object, using our symbol as the property key.

We need a proxy with set trap to call handlers in case of any change.

```

let handlers = Symbol('handlers');

function makeObservable(target) {
  // 1. Initialize handlers store
  target[handlers] = [];

  // Store the handler function in array for future calls

```

```

target.observe = function(handler) {
  this[handlers].push(handler);
};

// 2. Create a proxy to handle changes
return new Proxy(target, {
  set(target, property, value, receiver) {
    let success = Reflect.set(...arguments); // forward the operation to object
    if (success) { // if there were no error while setting the property
      // call all handlers
      target[handlers].forEach(handler => handler(property, value));
    }
    return success;
  }
});
}

let user = {};

user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John";

```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/proxy","identifier":"Vproxy"}); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true; TutorialThe JavaScript languageMiscellaneous{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Miscellaneous","item":"https://javascript.info/js-misc"}]}September 25, 2019Eval: run a code stringThe built-in eval function allows to execute a string of code. The syntax is:


```
let result = eval(code);
```

For example:

```
let code = 'alert("Hello")';  
eval(code); // Hello
```

A string of code may be long, contain line breaks, function declarations, variables and so on.

The result of eval is the result of the last statement.

For example:

```
let value = eval('1+1');  
alert(value); // 2
```

```
    let value = eval('let i = 0; ++i');  
    alert(value); // 1
```

The eval'ed code is executed in the current lexical environment, so it can see outer variables:

```
    let a = 1;  
  
    function f() {  
        let a = 2;  
  
        eval('alert(a)'); // 2  
    }  
  
    f();
```

It can change outer variables as well:

```
    let x = 5;  
eval("x = 10");  
alert(x); // 10, value modified
```

In strict mode, eval has its own lexical environment. So functions and variables, declared inside eval, are not visible outside:

```
// reminder: 'use strict' is enabled in runnable examples by default  
  
eval("let x = 5; function f() {}");  
  
alert(typeof x); // undefined (no such variable)  
// function f is also not visible
```

Without use strict, eval doesn't have its own lexical environment, so we would see x and f outside.

Using "eval" In modern programming eval is used very sparingly. It's often said that "eval is evil".

The reason is simple: long, long time ago JavaScript was a much weaker language, many things could only be done with eval. But that time passed a decade ago.

Right now, there's almost no reason to use eval. If someone is using it, there's a good chance they can replace it with a modern language construct or a JavaScript Module. Please note that its ability to access outer variables has side-effects.

Code minifiers (tools used before JS gets to production, to compress it) rename local variables into shorter ones (like a, b etc) to make the code smaller. That's usually safe, but not if eval is used, as local variables may be accessed from eval'ed code string. So minifiers don't do that renaming for all variables potentially visible from eval. That negatively affects code compression ratio.

Using outer local variables inside eval is also considered a bad programming practice, as it makes maintaining the code more difficult.

There are two ways how to be totally safe from such problems.

If eval'ed code doesn't use outer variables, please call eval as window.eval(...):

This way the code is executed in the global scope:

```
    let x = 1;
{
  let x = 5;
  window.eval('alert(x)'); // 1 (global variable)
}
```

If eval'ed code needs local variables, change eval to new Function and pass them as arguments:

```
let f = new Function('a', 'alert(a)');  
  
f(5); // 5
```

The new Function construct is explained in the chapter The "new Function" syntax. It creates a function from a string, also in the global scope. So it can't see local variables. But it's so much clearer to pass them explicitly as arguments, like in the example above.

SummaryA call to eval(code) runs the string of code and returns the result of the last statement.

Rarely used in modern JavaScript, as there's usually no need.

Can access outer local variables. That's considered bad practice.

Instead, to eval the code in the global scope, use window.eval(code).

Or, if your code needs some data from the outer scope, use new Function and pass it as arguments.

TasksEval-calculatorimportance: 4Create a calculator that prompts for an arithmetic expression and returns its result.

There's no need to check the expression for correctness in this task. Just evaluate and return the result.

Run the demosolutionLet's use eval to calculate the maths expression:

```
let expr = prompt("Type an arithmetic expression?", '2*3+2');  
  
alert( eval(expr) );
```

The user can input any text or code though.

To make things safe, and limit it to arithmetics only, we can check the expr using a regular expression, so that it only may contain digits and operators.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https://javascript.info/eval","identifier":"Veal"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialThe JavaScript languageMiscellaneous{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"The JavaScript language","item":"https://javascript.info/js"}, {"@type":"ListItem","position":3,"name":"Miscellaneous","item":"https://javascript.info/js-misc"}]}

January 10, 2021CurryingCurrying is an advanced technique of working with functions. It's used not only in JavaScript, but in other languages as well. Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c). Currying doesn't call a function. It just transforms it. Let's see an example first, to better understand what we're talking about, and then practical applications. We'll create a helper function curry(f) that performs currying for a two-argument f. In other words, curry(f) for two-argument f(a, b) translates it into a function that runs as f(a)(b):

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// usage
function sum(a, b) {
```

```
    return a + b;
  }

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

As you can see, the implementation is straightforward: it's just two wrappers.

The result of `curry(func)` is a wrapper function(a).
When it is called like `curriedSum(1)`, the argument is saved in the Lexical Environment, and a new wrapper is returned function(b).
Then this wrapper is called with 2 as an argument, and it passes the call to the original `sum`.

More advanced implementations of currying, such as `_.curry` from lodash library, return a wrapper that allows a function to be called both normally and partially:

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // using _.curry from lodash library

alert( curriedSum(1, 2) ); // 3, still callable normally
alert( curriedSum(1)(2) ); // 3, called partially
```

Currying? What for? To understand the benefits we need a worthy real-life example. For instance, we have the logging function `log(date, importance, message)` that formats and outputs the information. In real projects such functions have many useful features like sending logs over the network, here we'll just use `alert`:

```
function log(date, importance, message) {  
  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);  
}
```

Let's curry it!

```
log = _.curry(log);
```

After that log works normally:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

...But also works in the curried form:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Now we can easily make a convenience function for current logs:

```
// logNow will be the partial of log with fixed first argument  
let logNow = log(new Date());  
  
// use it
```



```
logNow("INFO", "message"); // [HH:mm] INFO message
```

Now `logNow` is `log` with fixed first argument, in other words “partially applied function” or “partial” for short.

We can go further and make a convenience function for current debug logs:

```
let debugNow = logNow("DEBUG");  
  
debugNow("message"); // [HH:mm] DEBUG message
```

So:

We didn’t lose anything after currying: `log` is still callable normally.
We can easily generate partial functions such as for today’s logs.

Advanced curry implementation
In case you’d like to get in to the details, here’s the “advanced” curry implementation for multi-argument functions that we could use above.
It’s pretty short:

```
function curry(func) {  
  return function curried(...args) {  
    if (args.length >= func.length) {  
      return func.apply(this, args);  
    } else {  
      return function(...args2) {  
        return curried.apply(this, args.concat(args2));  
      }  
    }  
  };  
}
```

Usage examples:

```

    function sum(a, b, c) {
    return a + b + c;
    }

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying

```

The new curry may look complicated, but it's actually easy to understand. The result of `curry(func)` call is the wrapper `curried` that looks like this:

```

    // func is the function to transform
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};

```

When we run it, there are two if execution branches:

If passed args count is the same or more than the original function has in its definition (`func.length`), then just pass the call to it using `func.apply`. Otherwise, get a partial: we don't call `func` just yet. Instead, another wrapper is returned, that will re-apply `curried` providing previous arguments together with the new ones.

Then, if we call it, again, we'll get either a new partial (if not enough arguments) or, finally, the result.

Fixed-length functions only

The currying requires the function to have a fixed number of arguments.
A function that uses rest parameters, such as `f(...args)`, can't be curried this way.

A little more than currying

By definition, currying should convert `sum(a, b, c)` into `sum(a)(b)(c)`.
But most implementations of currying in JavaScript are advanced, as described: they also keep the function callable in the multi-argument variant.

SummaryCurrying is a transform that makes `f(a,b,c)` callable as `f(a)(b)(c)`. JavaScript implementations usually both keep the function callable normally and return the partial if the arguments count is not enough.

Currying allows us to easily get partials. As we've seen in the logging example, after currying the three argument universal function `log(date, importance, message)` gives us partials when called with one argument (like `log(date)`) or two arguments (like `log(date, importance)`).

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {};
Object.assign(this.page, { "url": "https://javascript.info/currying-partials", "identifier": "Vcurrying-partials" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;
TutorialThe JavaScript languageMiscellaneous{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"},
{"@type": "ListItem", "position": 3, "name": "Miscellaneous", "item": "https://javascript.info/js-misc"}]}}September 21, 2022Reference Type

In-depth language feature

This article covers an advanced topic, to understand certain edge-cases better. It's not important. Many experienced developers live fine without knowing it. Read on if you want to know how things work under the hood.

A dynamically evaluated method call can lose this.
For instance:

```
    let user = {  
    name: "John",  
    hi() { alert(this.name); },  
    bye() { alert("Bye"); }  
};
```

```
user.hi(); // works
```

```
// now let's call user.hi or user.bye depending on the name  
(user.name == "John" ? user.hi : user.bye()); // Error!
```

On the last line there is a conditional operator that chooses either `user.hi` or `user.bye`. In this case the result is `user.hi`. Then the method is immediately called with parentheses `()`. But it doesn't work correctly! As you can see, the call results in an error, because the value of `"this"` inside the call becomes undefined.
This works (object dot method):

```
user.hi();
```

This doesn't (evaluated method):

```
(user.name == "John" ? user.hi : user.bye()); // Error!
```

Why? If we want to understand why it happens, let's get under the hood of how `obj.method()` call works. Reference type explained Looking closely, we may notice two operations in `obj.method()` statement:

First, the dot '.' retrieves the property obj.method.
Then parentheses () execute it.

So, how does the information about this get passed from the first part to the second one?

If we put these operations on separate lines, then this will be lost for sure:

```
    let user = {  
      name: "John",  
      hi() { alert(this.name); }  
    };  
  
// split getting and calling the method in two lines  
let hi = user.hi;  
hi(); // Error, because this is undefined
```

Here hi = user.hi puts the function into the variable, and then on the last line it is completely standalone, and so there's no this.

To make user.hi() calls work, JavaScript uses a trick – the dot '.' returns not a function, but a value of the special Reference Type.

The Reference Type is a “specification type”. We can't explicitly use it, but it is used internally by the language.

The value of Reference Type is a three-value combination (base, name, strict), where:

base is the object.

name is the property name.

strict is true if use strict is in effect.

The result of a property access user.hi is not a function, but a value of Reference Type. For user.hi in strict mode it is:

```
// Reference Type value
(user, "hi", true)
```

When parentheses () are called on the Reference Type, they receive the full information about the object and its method, and can set the right this (user in this case).

Reference type is a special “intermediary” internal type, with the purpose to pass information from dot . to calling parentheses ().

Any other operation like assignment `hi = user.hi` discards the reference type as a whole, takes the value of `user.hi` (a function) and passes it on. So any further operation “loses” this.

So, as the result, the value of this is only passed the right way if the function is called directly using a dot `obj.method()` or square brackets `obj['method']()` syntax (they do the same here). There are various ways to solve this problem such as `func.bind()`.

SummaryReference Type is an internal type of the language.

Reading a property, such as with dot . in `obj.method()` returns not exactly the property value, but a special “reference type” value that stores both the property value and the object it was taken from.

That’s for the subsequent method call () to get the object and set this to it.

For all other operations, the reference type automatically becomes the property value (a function in our case).

The whole mechanics is hidden from our eyes. It only matters in subtle cases, such as when a method is obtained dynamically from the object, using an expression.

TasksSyntax checkimportance: 2What is the result of this code?

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.S. There’s a pitfall :)

solutionError!

Try it:

```
let user = {  
  name: "John",  
  go: function() { alert(this.name) }  
}  
  
(user.go)() // error!
```

The error message in most browsers does not give us much of a clue about what went wrong.

The error appears because a semicolon is missing after `user = {...}`.

JavaScript does not auto-insert a semicolon before a bracket `(user.go)()`, so it reads the code like:

```
let user = { go:... }(user.go)()
```

Then we can also see that such a joint expression is syntactically a call of the object `{ go: ... }` as a function with the argument `(user.go)`. And that also happens on the same line with `let user`, so the `user` object has not yet even been defined, hence the error.

If we insert the semicolon, all is fine:

```

    let user = {
      name: "John",
      go: function() { alert(this.name) }
    };

```

```

(user.go)() // John

```

Please note that parentheses around (user.go) do nothing here. Usually they setup the order of operations, but here the dot . works first anyway, so there's no effect. Only the semicolon thing matters.

Explain the value of "this" importance: 3In the code below we intend to call obj.go() method 4 times in a row.

But calls (1) and (2) works differently from (3) and (4). Why?

```

    let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();           // (1) [object Object]

(obj.go)();         // (2) [object Object]

(method = obj.go)(); // (3) undefined

(obj.go || obj.stop)(); // (4) undefined

```

solutionHere's the explanations.

That's a regular object method call.

The same, parentheses do not change the order of operations here, the dot is first anyway.

Here we have a more complex call (expression)(). The call works as if it were split into two lines:

```
f = obj.go; // calculate the expression
f();      // call what we have
```

Here f() is executed as a function, without this.

The similar thing as (3), to the left of the parentheses () we have an expression.

To explain the behavior of (3) and (4) we need to recall that property accessors (dot or square brackets) return a value of the Reference Type.

Any operation on it except a method call (like assignment = or ||) turns it into an ordinary value, which does not carry the information allowing to set this.

Ctrl + !•Previous lessonCtrl + !Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article

– please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)

```
var DisqusConfig = function() { if (!this.page) this.page = {};
```

```
Object.assign(this.page, { "url": "https://javascript.info/reference-type", "identifier": "reference-type" }); }; var DisqusShortname = "javascriptinfo"; var DisqusEnabled = true;
```

```
TutorialThe JavaScript languageMiscellaneous{"@context": "https://schema.org", "@type": "BreadcrumbList", "itemListElement":
```

```
[{"@type": "ListItem", "position": 1, "name": "Tutorial", "item": "https://javascript.info/"},
```

```
{"@type": "ListItem", "position": 2, "name": "The JavaScript language", "item": "https://javascript.info/js"},
```

```
{"@type": "ListItem", "position": 3, "name": "Miscellaneous", "item": "https://javascript.info/js-misc"}]]September 21, 2020BigInt
```

A recent addition

This is a recent addition to the language.

You can find the current state of support at <https://caniuse.com/#feat=bigint>.

BigInt is a special numeric type that provides support for integers of arbitrary length. A bigint is created by appending n to the end of an integer literal or by calling the function BigInt that creates bigints from strings, numbers etc.

```
const bigint = 1234567890123456789012345678901234567890n;  
const sameBigInt = BigInt("1234567890123456789012345678901234567890");  
const bigintFromNumber = BigInt(10); // same as 10n
```

Math operators BigInt can mostly be used like a regular number, for example:

```
alert(1n + 2n); // 3  
alert(5n / 2n); // 2
```

Please note: the division 5/2 returns the result rounded towards zero, without the decimal part. All operations on bigints return bigints. We can't mix bigints and regular numbers:

```
alert(1n + 2); // Error: Cannot mix BigInt and other types
```

We should explicitly convert them if needed: using either `BigInt()` or `Number()`, like this:

```
let bigint = 1n;
let number = 2;

// number to bigint
alert(bigint + BigInt(number)); // 3

// bigint to number
alert(Number(bigint) + number); // 3
```

The conversion operations are always silent, never give errors, but if the bigint is too huge and won't fit the number type, then extra bits will be cut off, so we should be careful doing such conversion.

The unary plus is not supported on bigints

The unary plus operator `+value` is a well-known way to convert value to a number.

In order to avoid confusion, it's not supported on bigints:

```
let bigint = 1n;  
alert( +bigint ); // error
```

So we should use `Number()` to convert a bigint to a number.

Comparisons, such as `<`, `>` work with bigints and numbers just fine:

```
alert( 2n > 1n ); // true  
alert( 2n > 1 ); // true
```

Please note though, as numbers and bigints belong to different types, they can be equal `==`, but not strictly equal `===`:

```
alert( 1 == 1n ); // true
```

```
alert( 1 === 1n ); // false
```

Boolean operations When inside if or other boolean operations, bigints behave like numbers.

For instance, in if, bigint 0n is falsy, other values are truthy:

```
    if (0n) {  
    // never executes  
    }
```

Boolean operators, such as ||, && and others also work with bigints similar to numbers:

```
alert( 1n || 2 ); // 1 (1n is considered truthy)
```

```
alert( 0n || 2 ); // 2 (0n is considered falsy)
```

PolyfillsPolyfilling bigints is tricky. The reason is that many JavaScript operators, such as +, - and so on behave differently with bigints compared to regular numbers. For example, division of bigints always returns a bigint (rounded if necessary).

To emulate such behavior, a polyfill would need to analyze the code and replace all such operators with its functions. But doing so is cumbersome and would cost a lot of performance.

So, there's no well-known good polyfill.

Although, the other way around is proposed by the developers of JSBI library.

This library implements big numbers using its own methods. We can use them instead of native bigints:

Operation
native BigInt
JSBI

Creation from Number
a = BigInt(789)
a = JSBI.BigInt(789)

Addition
c = a + b
c = JSBI.add(a, b)

Subtraction
c = a - b
c = JSBI.subtract(a, b)

...
...
...

...And then use the polyfill (Babel plugin) to convert JSBI calls to native bigints for those browsers that support them.

In other words, this approach suggests that we write code in JSBI instead of native bigints. But JSBI works with numbers as with bigints internally, emulates them closely following the specification, so the code will be “bigint-ready”.

We can use such JSBI code “as is” for engines that don’t support bigints and for those that do support – the polyfill will convert the calls to native bigints.

References

MDN docs on BigInt.

Specification.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting...If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen...)var DisqusConfig = function() { if (!this.page) this.page = {}; Object.assign(this.page, { "url": "https://javascript.info/bigint", "identifier": "Vbigint" }); };var DisqusShortname = "javascriptinfo";var DisqusEnabled = true;