

Part 1

The JavaScript language

JS

Ilya Kantor

Built at November 26, 2018

The last version of the tutorial is at <https://javascript.info>.

We constantly work to improve the tutorial. If you find any mistakes, please write at [our github ↗](#).

- [An introduction](#)
 - [An Introduction to JavaScript](#)
 - [Code editors](#)
 - [Developer console](#)
- [JavaScript Fundamentals](#)
 - [Hello, world!](#)
 - [Code structure](#)
 - [The modern mode, "use strict"](#)
 - [Variables](#)
 - [Data types](#)
 - [Type Conversions](#)
 - [Operators](#)
 - [Comparisons](#)
 - [Interaction: alert, prompt, confirm](#)
 - [Conditional operators: if, ?:](#)
 - [Logical operators](#)
 - [Loops: while and for](#)
 - [The "switch" statement](#)
 - [Functions](#)
 - [Function expressions and arrows](#)
 - [JavaScript specials](#)
- [Code quality](#)
 - [Debugging in Chrome](#)
 - [Coding Style](#)
 - [Comments](#)
 - [Ninja code](#)
 - [Automated testing with mocha](#)
 - [Polyfills](#)
- [Objects: the basics](#)
 - [Objects](#)
 - [Garbage collection](#)
 - [Symbol type](#)
 - [Object methods, "this"](#)
 - [Object to primitive conversion](#)
 - [Constructor, operator "new"](#)
- [Data types](#)

- Methods of primitives
- Numbers
- Strings
- Arrays
- Array methods
- Iterables
- Map, Set, WeakMap and WeakSet
- Object.keys, values, entries
- Destructuring assignment
- Date and time
- JSON methods, toJSON
- Advanced working with functions
 - Recursion and stack
 - Rest parameters and spread operator
 - Closure
 - The old "var"
 - Global object
 - Function object, NFE
 - The "new Function" syntax
 - Scheduling: setTimeout and setInterval
 - Decorators and forwarding, call/apply
 - Function binding
 - Currying and partials
 - Arrow functions revisited
- Objects, classes, inheritance
 - Property flags and descriptors
 - Property getters and setters
 - Prototypal inheritance
 - F.prototype
 - Native prototypes
 - Methods for prototypes
 - Class patterns
 - Classes
 - Class inheritance, super
 - Class checking: "instanceof"
 - Mixins
- Error handling
 - Error handling, "try..catch"
 - Custom errors, extending Error

Here we learn JavaScript, starting from scratch and go on to advanced concepts like OOP. We concentrate on the language itself here, with the minimum of environment-specific notes.

An introduction

About the JavaScript language and the environment to develop with it.

An Introduction to JavaScript

Let's see what's so special about JavaScript, what we can achieve with it, and which 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 the HTML and executed automatically as the page loads.

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

In this aspect, JavaScript is very different from another language called [Java ↗](#).

Why JavaScript?

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

But as it evolved, JavaScript became a fully independent language, with its own specification called [ECMAScript ↗](#), and now it has no relation to Java at all.

At present, JavaScript can not only execute 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 and Opera.
- [SpiderMonkey ↗](#) – in Firefox.
- ... There are other codenames like “Trident” and “Chakra” for different versions of IE, “ChakraCore” for Microsoft Edge, “Nitro” and “SquirrelFish” for Safari, etc.

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

i How do engines work?

Engines are complicated. But the basics are easy.

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

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

What can in-browser JavaScript do?

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

The capabilities greatly depend on the environment that runs JavaScript. For instance, [Node.js](#) 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 for the sake of the user's safety. The aim is to prevent an evil webpage from accessing private information or harming the user's data.

The examples of such restrictions are:

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

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

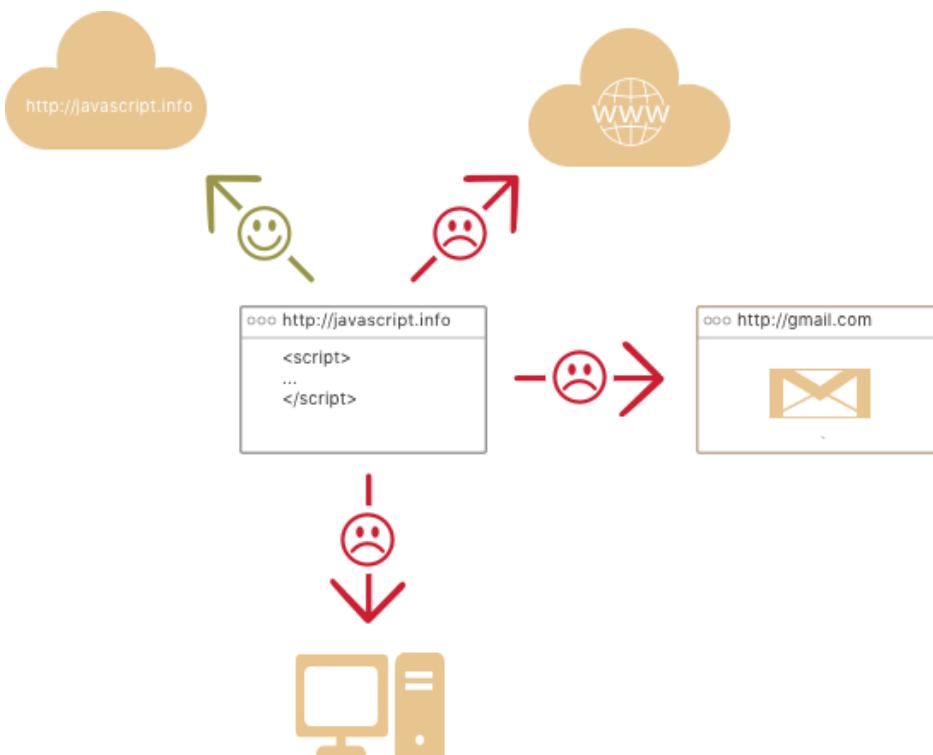
There are ways to interact with camera/microphone and other devices, but they require a user's explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the [NSA](#).

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

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

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

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



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

What makes JavaScript unique?

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.

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

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

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

Languages “over” JavaScript

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 a “syntactic sugar” for JavaScript, it introduces shorter syntax, allowing to write more precise and clear 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.
- [Dart ↗](#) is a standalone language that has its own engine that runs in non-browser environments (like mobile apps). It was initially offered by Google as a replacement for JavaScript, but as of now, browsers require it to be transpiled to JavaScript just like the ones above.

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

Summary

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

Code editors

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

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

IDE

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

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

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

- [WebStorm ↗](#) for frontend development and other editors of the same company if you need additional languages (paid).
- [Visual Studio Code ↗](#) (free).
- [Netbeans ↗](#) (paid).

All of the IDEs are cross-platform.

For Windows, there’s also a “Visual Studio” editor, don’t confuse it with “Visual Studio Code.” “Visual Studio” is a paid and mighty Windows-only editor, well-suited for the .NET platform. A free version of it is called ([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.

The following options deserve your attention:

- [Visual Studio Code ↗](#) (cross-platform, free).
- [Atom ↗](#) (cross-platform, free).
- [Sublime Text ↗](#) (cross-platform, shareware).
- [Notepad++ ↗](#) (Windows, free).
- [Vim ↗](#) and [Emacs ↗](#) are also cool if you know how to use them.

My favorites

The personal preference of the author is to have both an IDE for projects and a lightweight editor for quick and easy file editing.

I'm using:

- [WebStorm ↗](#) for JS, and if there is one more language in the project, then I switch to one of the other JetBrains offerings listed above.
- As a lightweight editor – [Sublime Text ↗](#) or [Atom ↗](#).

Let's not argue

The 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, personal preferences.

Developer console

Code is prone to errors. You are quite likely to make errors... Oh, what am I talking about? You are *absolutely* going to make errors, at least if you're a human, not a [robot ↗](#).

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

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

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

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

Google Chrome

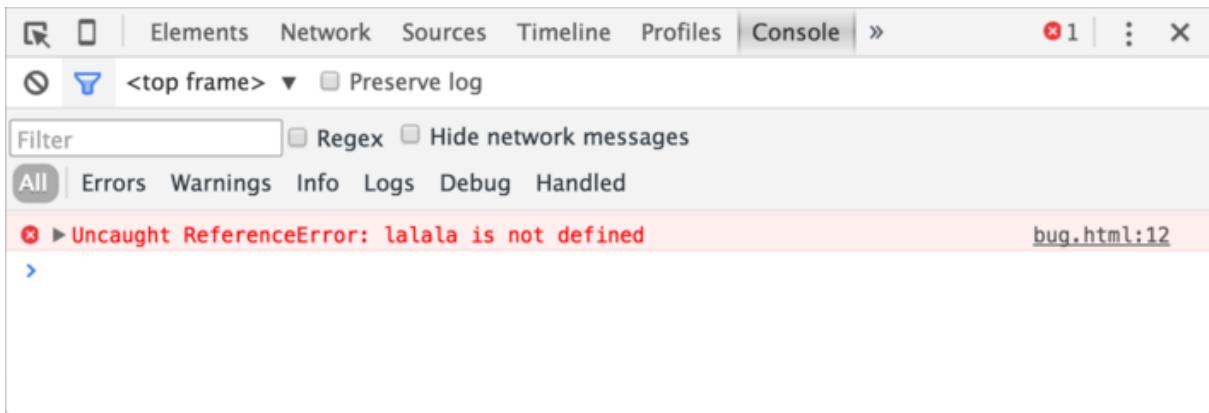
Open the page [bug.html](#).

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 (`Shift+Enter` to input multi-line commands).

Now we can see errors, and that’s enough for a start. We’ll be back to developer tools later and cover debugging more in-depth in the chapter [Debugging in Chrome](#).

Firefox, Edge, and others

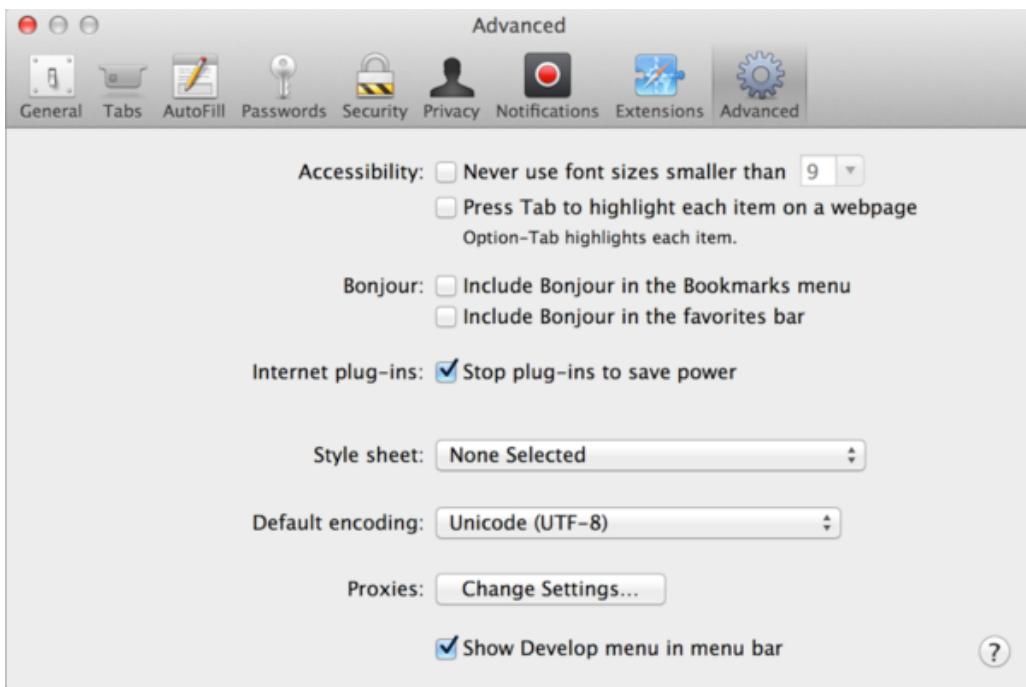
Most other browsers use `F12` to open developer tools.

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

Safari

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

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



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

Summary

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

Now we have the environment ready. In the next section, we'll get down to JavaScript.

JavaScript Fundamentals

Let's learn the fundamentals of script building.

Hello, world!

The tutorial that you're reading is about core JavaScript, which is platform-independent. Further on, you will learn Node.JS and other platforms that use it.

But, we need a working environment to run our scripts, and, just because 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. On the other hand, browser details are explained in detail in the [next part](#) of the tutorial.

So first, let's see how to attach a script to a webpage. For server-side environments, you can just execute it with a command like "`node my.js`" for Node.JS.

The "script" tag

JavaScript programs can be inserted in any part of an HTML document with the help of 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>
```

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

The modern markup

The `<script>` tag has a few attributes that are rarely used nowadays, but we can find them in old code:

The `type` attribute: `<script type=...>`

The old standard HTML4 required a script to have a type. Usually it was `type="text/javascript"`. It's not required anymore. Also, the modern 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 later 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. No need to use it.

Comments before and after scripts.

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

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

This trick isn't used in modern JavaScript. These comments were used to hide the JavaScript code from old browsers that didn't know about a `<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.

The script file is 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 file with the script (from the site root).

It is also possible to provide a path relative to the current page. For instance, `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/3.2.0/lodash.js"></script>
```

To attach several scripts, use multiple tags:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...

```

i 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 then store it in its [cache ↗](#).

After this, other pages that want the same script will take it from the cache instead of downloading it. So the file is actually downloaded only once.

That saves 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 the 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 it's 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 the 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 web-page. But let's keep in mind that this part of the tutorial is devoted to the JavaScript language, so we shouldn't distract ourselves from it. We'll be using a browser as a way to run JavaScript, which is very convenient for online reading, but yet one of many.

Tasks

Show an alert

importance: 5

Create a page that shows a message "I'm JavaScript!".

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

[Demo in new window ↗](#)

[To solution](#)

Show an alert with an external script

importance: 5

Take 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.

[To solution](#)

Code structure

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

Statements

Statements are syntax constructs and commands that perform actions.

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

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

For example, here we split the message into two:

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

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

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

Semicolons

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

This would also work:

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

Here JavaScript interprets the line break as an “implicit” semicolon. That's also called an [automatic semicolon insertion ↗](#).

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 the semicolon is not required. And in this case that works as intended.

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

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

An example of an error

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

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

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

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

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

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

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

```
alert("All fine now");  
[1, 2].forEach(alert)
```

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

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

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

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

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

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

Comments

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

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

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

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

Like here:

```
// This comment occupies a line of its own
alert('Hello');

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

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

Like this:

```
/* An example with two messages.
This is a multiline comment.
*/
alert('Hello');
alert('World');
```

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

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

```
/* Commenting out the code
alert('Hello');
*/
alert('World');
```

Use hotkeys!

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

Nested comments are not supported!

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

Such code will die with an error:

```
/*
  /* nested comment ?!?
*/
alert( 'World' );
```

Please, don't hesitate to comment your code.

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

Further in the tutorial there will be a chapter [Coding Style](#) that also explains how to write better comments.

The modern mode, "use strict"

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

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

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

"use strict"

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

For example

```
"use strict";  
  
// this code works the modern way  
...  
...
```

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

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

Ensure that “use strict” is at the top

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

There is no strict mode here:

```
alert("some code");
// "use strict" below is ignored, must be on the top

"use strict";

// strict mode is not activated
```

Only comments may appear above `"use strict"`.

There's no way to cancel `use strict`

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

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

Always “use strict”

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

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

At this point in time it's enough to know about it in general:

1. The `"use strict"` directive switches the engine to the “modern” mode, changing the behavior of some built-in features. We'll see the details as we study.
2. The strict mode is enabled by `"use strict"` at the top. Also there are several language features like “classes” and “modules” that enable strict mode automatically.
3. The strict mode is supported by all modern browsers.
4. It's always recommended to start scripts with `"use strict"`. All examples in this tutorial assume so, unless (very rarely) specified otherwise.

Variables

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

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

Variables are used to store this information.

A variable

A **variable ↗** is a “named storage” for data. We can use variables to store goodies, visitors and other data.

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

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

```
let message;
```

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

```
let message;  
  
message = 'Hello'; // store the string
```

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

```
let message;  
message = 'Hello!';  
  
alert(message); // shows the variable content
```

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

```
let message = 'Hello!'; // define the variable and assign the value  
  
alert(message); // Hello!
```

We can also declare multiple variables in one line:

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

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

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

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

Some people also write many variables like that:

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

...Or even in the “comma-first” style:

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

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

var instead of let

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

```
var message = 'Hello';
```

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

There are subtle differences between `let` and `var`, but they do not matter for us yet. We'll cover them in detail later, in the chapter [The old "var"](#).

A real-life analogy

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

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



We can put any value into the box.

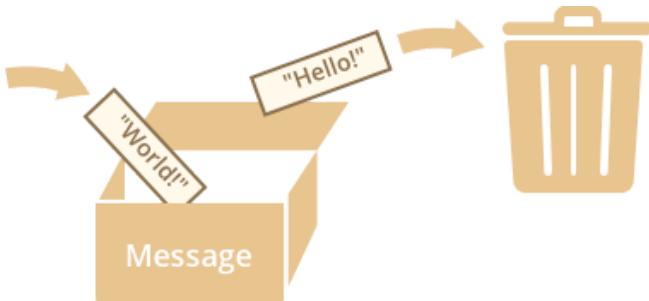
Also we can change it. The value can be changed as many times as needed:

```
let message;

message = 'Hello!';
```

```
message = 'World!'; // value changed  
alert(message);
```

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



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

```
let hello = 'Hello world!';  
  
let message;  
  
// copy 'Hello world' from hello into message  
message = hello;  
  
// now two variables hold the same data  
alert(hello); // Hello world!  
alert(message); // Hello world!
```

Functional languages

It may be interesting to know that there also exist [functional](#) programming languages that forbid changing a variable value. For example, [Scala](#) or [Erlang](#).

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

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

Variable naming

There are two limitations for a variable name in JavaScript:

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

Valid names, for instance:

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

When the name contains multiple words, [camelCase ↗](#) is commonly used. That is: words go one after another, each word starts with a capital letter: `myVeryLongName`.

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

These names are valid:

```
let $ = 1; // declared a variable with the name "$"  
let _ = 2; // and now a variable with the name "_"  
  
alert($ + _); // 3
```

Examples of incorrect variable names:

```
let 1a; // cannot start with a digit  
  
let my-name; // a hyphen '-' is not allowed in the name
```

➊ Case matters

Variables named `apple` and `AppLE` – are two different variables.

➋ Non-English letters are allowed, but not recommended

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

```
let имя = '...';  
let 我 = '...';
```

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

Reserved names

There is a [list of reserved words ↗](#), which cannot be used as variable names, because they are used by the language itself.

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

The code below gives a syntax error:

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

An assignment without `use strict`

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

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

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

alert(num); // 5
```

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

```
"use strict";

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

Constants

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

```
const myBirthday = '18.04.1982';
```

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

```
const myBirthday = '18.04.1982';

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

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

Like this:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Benefits:

- `COLOR_ORANGE` is much easier to remember than `"#FF7F00"`.
- It is much easier to mistype in `"#FF7F00"` than in `COLOR_ORANGE`.
- When reading the code, `COLOR_ORANGE` is much more meaningful than `#FF7F00`.

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

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

For instance:

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

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

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

Name things right

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

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

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

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

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

Some good-to-follow rules are:

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

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

Reuse or create?

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

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

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

An extra variable is good, not evil.

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

Summary

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

- `let` – is a modern variable declaration. The code must be in strict mode to use `let` in Chrome (V8).
- `var` – is an old-school variable declaration. Normally we don't use it at all, but we'll cover subtle differences from `let` in the chapter [The old "var"](#), 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.

Tasks

Working with variables

importance: 2

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

[To solution](#)

Giving the right name

importance: 3

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

[To solution](#)

Uppercase const?

importance: 4

Examine the following code:

```
const birthday = '18.04.1982';  
  
const age = someCode(birthday);
```

Here we have a constant `birthday` date and the `age` is calculated from `birthday` with the help of some code (it is not provided for shortness, and because details don't matter here).

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

```
const BIRTHDAY = '18.04.1982'; // make uppercase?  
  
const AGE = someCode(BIRTHDAY); // make uppercase?
```

[To solution](#)

Data types

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

```
// no error
let message = "hello";
message = 123456;
```

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

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

A number

```
let n = 123;
n = 12.345;
```

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

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

Besides regular numbers, there are so-called “special numeric values” which also belong to that type: `Infinity`, `-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 mention it in the code directly:

```
alert( Infinity ); // Infinity
```

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

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

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

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

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

Mathematical operations are safe

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

The script will never stop with a fatal error ("die"). At worst we'll get `NaN` as the result.

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

We'll see more about working with numbers in the chapter [Numbers](#).

A string

A string in JavaScript must be quoted.

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed ${str}`;
```

In JavaScript, there are 3 types of quotes.

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

Double and single quotes are "simple" quotes. There's no difference between them in JavaScript.

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

```
let name = "John";

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

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

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

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

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

We'll cover strings more thoroughly in the chapter [Strings](#).

i There is no **character** type.

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

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

A boolean (logical type)

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

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

For instance:

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

Boolean values also come as a result of comparisons:

```
let isGreater = 4 > 1;

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

We'll cover booleans more deeply later in the chapter [Logical operators](#).

The “null” value

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

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

```
let age = null;
```

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

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

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

The “undefined” value

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

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

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

```
let x;  
  
alert(x); // shows "undefined"
```

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

```
let x = 123;  
  
x = undefined;  
  
alert(x); // "undefined"
```

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

Objects and Symbols

The `object` type is special.

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

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

The `typeof` operator

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

It supports two forms of syntax:

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

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

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

```
typeof undefined // "undefined"  
typeof 0 // "number"  
typeof true // "boolean"  
typeof "foo" // "string"  
typeof Symbol("id") // "symbol"  
typeof Math // "object" (1)  
typeof null // "object" (2)  
typeof alert // "function" (3)
```

The last three lines may need additional explanations:

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

Summary

There are 7 basic types in JavaScript.

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

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

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

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

✓ Tasks

String quotes

importance: 5

What is the output of the script?

```
let name = "Ilya";

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

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

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

[To solution](#)

Type Conversions

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

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

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

i Not talking about objects yet

In this chapter we don't cover objects yet. Here we study primitives first. Later, after we learn objects, we'll see how object conversion works in the chapter [Object to primitive conversion](#).

ToString

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

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

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

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

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

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

ToNumber

Numeric conversion happens in mathematical functions and expressions automatically.

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

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

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

```
let str = "123";
alert(typeof str); // string

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

alert(typeof num); // number
```

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

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

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

alert(age); // NaN, conversion failed
```

Numeric conversion rules:

Value	Becomes...
<code>undefined</code>	<code>Nan</code>
<code>null</code>	<code>0</code>
<code>true</code> and <code>false</code>	<code>1</code> and <code>0</code>
<code>string</code>	Whitespaces from the start and the end are removed. Then, if the remaining string is empty, the result is <code>0</code> . Otherwise, the number is "read" from the string. An error gives <code>Nan</code> .

Examples:

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

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

Addition '+' concatenates strings

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

Then it concatenates (joins) them:

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

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

ToBoolean

Boolean conversion is the simplest one.

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

The conversion rule:

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

For instance:

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

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

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

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

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

Summary

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

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

ToNumber – Occurs in math operations, can be performed with `Number(value)`.

The conversion follows the rules:

Value	Becomes...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true / false</code>	<code>1 / 0</code>
<code>string</code>	The string is read "as is", whitespaces from both sides are ignored. An empty string becomes <code>0</code> . An error gives <code>NaN</code> .

ToBoolean – Occurs in logical operations, or can be performed with `Boolean(value)`.

Follows the rules:

Value	Becomes...
<code>0, null, undefined, NaN, ""</code>	<code>false</code>
any other value	<code>true</code>

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

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

Objects are not covered here, we'll return to them later in the chapter [Object to primitive conversion](#) that is devoted exclusively to objects, after we learn more basic things about JavaScript.

Tasks

Type conversions

importance: 5

What are results of these expressions?

```
"" + 1 + 0  
"" - 1 + 0  
true + false  
6 / "3"  
"2" * "3"
```

```
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
7 / 0
" -9\n" + 5
" -9\n" - 5
null + 1
undefined + 1
```

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

[To solution](#)

Operators

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

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

Terms: “unary”, “binary”, “operand”

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

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

```
let x = 1;

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

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

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

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

Strings concatenation, binary `+`

Now let's see special features of JavaScript operators that are beyond school arithmetics.

Usually the plus operator `+` sums numbers.

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

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

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

For example:

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

See, it doesn't matter whether the first operand is a string or the second one. The rule is simple: if either operand is a string, then convert the other one into a string as well.

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

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

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

For instance, subtraction and division:

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

Numeric conversion, unary `+`

The plus `+` exists in two forms. The binary form that we used above and the unary form.

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

For example:

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

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

// Converts non-numbers
```

```
alert( +true ); // 1
alert( +""); // 0
```

It actually does the same as `Number(...)`, but is shorter.

A need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, then they are usually strings.

What if we want to sum them?

The binary plus would add them as strings:

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

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

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

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

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

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

From a mathematician's standpoint the abundance of pluses may seem strange. But from a programmer's standpoint, there's nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up.

Why are unary pluses applied to values before the binary one? As we're going to see, that's because of their *higher precedence*.

Operators precedence

If an expression has more than one operator, the execution order is defined by their *precedence*, or, in other words, there's an implicit priority order among the operators.

From school we all know that the multiplication in the expression `1 + 2 * 2` should be calculated before the addition. That's exactly the precedence thing. The multiplication is said to have a *higher precedence* than the addition.

Parentheses override any precedence, so if we're not satisfied with the order, we can use them, like: `(1 + 2) * 2`.

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

An extract from the [precedence table ↗](#) (you don't need to remember this, but note that unary operators are higher than corresponding binary ones):

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

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

Assignment

Let's note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 3.

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

```
let x = 2 * 2 + 1;  
alert( x ); // 5
```

It is possible to chain assignments:

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

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

i The assignment operator "`=`" returns a value

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

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

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

```
let a = 1;  
let b = 2;  
  
let c = 3 - (a = b + 1);  
  
alert( a ); // 3  
alert( c ); // 0
```

In the example above, the result of `(a = b + 1)` is the value which is assigned to `a` (that is `3`). It is then used to subtract from `3`.

Funny code, isn't it? We should understand how it works, because sometimes we can see it in 3rd-party libraries, but shouldn't write anything like that ourselves. Such tricks definitely don't make the code clearer and readable.

Remainder %

The remainder operator `%` despite its look does not have a relation to percents.

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

For instance:

```
alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2  
alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3  
alert( 6 % 3 ); // 0 is a remainder of 6 divided by 3
```

Exponentiation **

The exponentiation operator `**` is a recent addition to the language.

For a natural number `b`, the result of `a ** b` is `a` multiplied by itself `b` times.

For instance:

```
alert( 2 ** 2 ); // 4 (2 * 2)  
alert( 2 ** 3 ); // 8 (2 * 2 * 2)  
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

The operator works for non-integer numbers of `a` and `b` as well, for instance:

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

Increment/decrement

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

So, there are special operators for that:

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

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

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

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

Important:

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

Operators `++` and `--` can be placed both after and before the variable.

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

Both of these records do the same: increase `counter` by 1.

Is there any difference? Yes, but we can only see it if we use the returned value of `++/- -`.

Let's clarify. As we know, all operators return a value. Increment/decrement is not an exception here. The prefix form returns the new value, while the postfix form returns the old value (prior to increment/decrement).

To see the difference, here's the example:

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

Here in the line `(*)` the prefix call `++counter` increments `counter` and returns the new value that is 2. So the `alert` shows 2.

Now let's use the postfix form:

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

alert(a); // 1
```

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

To summarize:

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

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

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

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

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

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

i Increment/decrement among other operators

Operators `++/-` can be used inside an expression as well. Their precedence is higher than most other arithmetical operations.

For instance:

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

Compare with:

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

Though technically allowable, such notation usually makes the code less readable. One line does multiple things – not good.

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

The “one line – one action” style is advised:

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

Bitwise operators

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

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

The list of operators:

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

These operators are used very rarely. To understand them, we should delve into low-level number representation, and it would not be optimal to do that right now. Especially because we won't need them any time soon. If you're curious, you can read the [Bitwise Operators](#) article in MDN. It would be more practical to do that when a real need arises.

Modify-in-place

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

For example:

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

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

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

Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: `/=`, `-=` etc.

Such operators have the same precedence as a normal assignment, so they run after most other calculations:

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

Comma

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

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

For example:

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

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

Comma has a very low precedence

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

Without them: `a = 1 + 2, 3 + 4` evaluates `+` first, summing the numbers into `a = 3, 7`, then the assignment operator `=` assigns `a = 3`, and then the number after the comma `7` is not processed anyhow, so it's ignored.

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

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

For example:

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

Such tricks are used in many JavaScript frameworks, that's why we mention them. But usually they don't improve the code readability, so we should think well before writing like that.

Tasks

The postfix and prefix forms

importance: 5

What are the final values of all variables `a`, `b`, `c` and `d` after the code below?

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

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

[To solution](#)

Assignment result

importance: 3

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

```
let a = 2;

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

[To solution](#)

Comparisons

Many comparison operators we know from maths:

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

Boolean is the result

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

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

For example:

```
alert( 2 > 1 ); // true (correct)
alert( 2 == 1 ); // false (wrong)
alert( 2 != 1 ); // true (correct)
```

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

```
let result = 5 > 4; // assign the result of the comparison
alert( result ); // true
```

String comparison

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

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

For example:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

The algorithm to compare two strings is simple:

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

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

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

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

Not a real dictionary, but Unicode order

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

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

Comparison of different types

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

For example:

```
alert( '2' > 1 ); // true, string '2' becomes a number 2
alert( '01' == 1 ); // true, string '01' becomes a number 1
```

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

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

A funny consequence

It is possible that at the same time:

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

For example:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

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

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

Strict equality

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

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

The same thing with an empty string:

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

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

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

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

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

Let's try it:

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

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

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

Comparison with null and undefined

Let's see more edge cases.

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

For a strict equality check `==`

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

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

For a non-strict check `=`

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

```
alert( null == undefined ); // true
```

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

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

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

Strange result: null vs 0

Let's compare `null` with a zero:

```
alert( null > 0 ); // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

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

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

On the other hand, the equality check `==` for `undefined` and `null` 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 participate in comparisons at all:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

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

We've got these results because:

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

Evade problems

Why did we observe these examples? Should we remember these peculiarities all the time?

Well, not really. Actually, these tricky things will gradually become familiar over time, but there's a solid way to evade any problems with them.

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

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

Summary

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

Tasks

Comparisons

importance: 5

What will be the result for expressions?

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\\n0\\n"
```

[To solution](#)

Interaction: alert, prompt, confirm

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

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

alert

Syntax:

```
alert(message);
```

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

For example:

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

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

prompt

Function `prompt` accepts two arguments:

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

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

title

The text to show to the visitor.

default

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

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

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

For instance:

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

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

⚠ IE: always supply a `default`

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

Run this code in Internet Explorer to see that:

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

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

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

confirm

The syntax:

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

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

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

For example:

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

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

Summary

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

`alert`

shows a message.

prompt

shows a message asking the user to input text. It returns the text or, if CANCEL or `Esc` is clicked, all browsers return `null`.

confirm

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

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

There are two limitations shared by all the methods above:

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

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

Tasks

A simple page

importance: 4

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

[Run the demo](#)

[To solution](#)

Conditional operators: if, '?'

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

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

The “if” statement

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

For example:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');
if (year == 2015) alert( 'You are right!' );
```

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

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

```
if (year == 2015) {  
    alert( "That's correct!" );  
    alert( "You're so smart!" );  
}
```

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

Boolean conversion

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

Let's recall the conversion rules from the chapter [Type Conversions](#):

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

So, the code under this condition would never execute:

```
if (0) { // 0 is falsy  
    ...  
}
```

...And inside this condition – always works:

```
if (1) { // 1 is truthy  
    ...  
}
```

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

```
let cond = (year == 2015); // equality evaluates to true or false  
  
if (cond) {  
    ...  
}
```

The “else” clause

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

For example:

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

if (year == 2015) {
  alert( 'You guessed it right!' );
} else {
  alert( 'How can you be so wrong?' ); // any value except 2015
}
```

Several conditions: “else if”

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

For example:

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

if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
  alert( 'Too late' );
} else {
  alert( 'Exactly!' );
}
```

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

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

Ternary operator ‘?’

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

For instance:

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

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
```

```
alert(accessAllowed);
```

The so-called “ternary” or “question mark” operator lets us do that shorter and simpler.

The operator is represented by a question mark `?`. The formal term “ternary” means that the operator has three operands. It is actually the one and only operator in JavaScript which has that many.

The syntax is:

```
let result = condition ? value1 : value2
```

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

For example:

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

Technically, we can omit parentheses around `age > 18`. The question mark operator has a low precedence. It executes after the comparison `>`, so that'll do the same:

```
// the comparison operator "age > 18" executes first anyway
// (no need to wrap it into parentheses)
let accessAllowed = age > 18 ? true : false;
```

But parentheses make the code more readable, so it's recommended to use them.

i Please note:

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

```
// the same
let accessAllowed = age > 18;
```

Multiple ‘?’

A sequence of question mark `?` operators allows returning a value that depends on more than one condition.

For instance:

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

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

```

(age < 18) ? 'Hello!' :
(age < 100) ? 'Greetings!' :
'What an unusual age!';

alert( message );

```

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

1. The first question mark checks whether `age < 3`.
2. If true – returns `'Hi, baby!'`, otherwise – goes after the colon :" and checks for `age < 18`.
3. If that's true – returns `'Hello!'`, otherwise – goes after the next colon :" and checks for `age < 100`.
4. If that's true – returns `'Greetings!'`, otherwise – goes after the last colon :" and returns `'What an unusual age!'`.

The same logic using `if..else`:

```

if (age < 3) {
  message = 'Hi, baby!';
} else if (age < 18) {
  message = 'Hello!';
} else if (age < 100) {
  message = 'Greetings!';
} else {
  message = 'What an unusual age!';
}

```

Non-traditional use of ‘?’

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

```

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

(company == 'Netscape') ?
  alert('Right!') : alert('Wrong.');

```

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

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

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

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

Here is the same code with `if` for comparison:

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

if (company == 'Netscape') {
    alert('Right!');
} else {
    alert('Wrong.');
}
```

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

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

✓ Tasks

if (a string with zero)

importance: 5

Will `alert` be shown?

```
if ("0") {
    alert( 'Hello' );
}
```

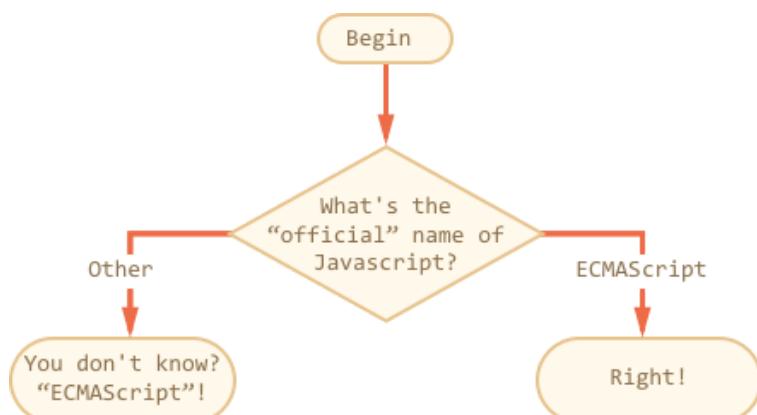
[To solution](#)

The name of JavaScript

importance: 2

Using the `if..else` construct, write the code which asks: 'What is the "official" name of JavaScript?'

If the visitor enters "ECMAScript", then output "Right!", otherwise – output: "Didn't know? ECMAScript!"



[Demo in new window ↗](#)

[To solution](#)

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 ↗](#)

[To solution](#)

Check the login

importance: 3

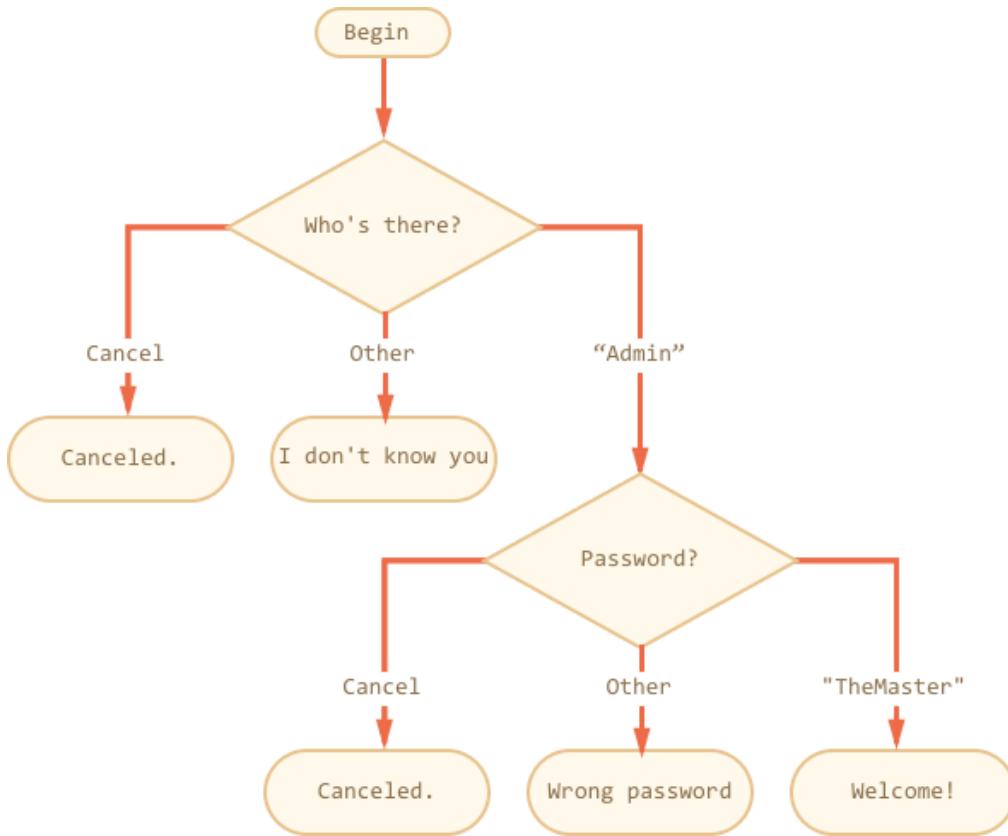
Write the code which asks for a login with `prompt`.

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

The password is checked as follows:

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

The schema:



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

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

[Run the demo](#)

[To solution](#)

Rewrite 'if' into '?"

importance: 5

Rewrite this `if` using the ternary operator `'?'`:

```

if (a + b < 4) {
    result = 'Below';
} else {
    result = 'Over';
}

```

[To solution](#)

Rewrite 'if..else' into '?"

importance: 5

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

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

```
let message;

if (login == 'Employee') {
    message = 'Hello';
} else if (login == 'Director') {
    message = 'Greetings';
} else if (login == '') {
    message = 'No login';
} else {
    message = '';
}
```

[To solution](#)

Logical operators

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

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

Let's see the details.

`|| (OR)`

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

```
result = a || b;
```

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

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

There are four possible logical combinations:

```
alert( true || true ); // true
alert( false || true ); // true
alert( true || false ); // true
alert( false || false ); // false
```

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

If an operand is not boolean, then it's converted to boolean for the evaluation.

For instance, a number `1` is treated as `true`, a number `0` – as `false`:

```
if (1 || 0) { // works just like if( true || false )
  alert( 'truthy!' );
}
```

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

For example:

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'The office is closed.' );
}
```

We can pass more conditions:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'The office is closed.' ); // it is the weekend
}
```

OR seeks the first truthy value

The logic described above is somewhat classical. Now let's bring in the "extra" features of JavaScript.

The extended algorithm works as follows.

Given multiple OR'ed values:

```
result = value1 || value2 || value3;
```

The OR `||` operator does the following:

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

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

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

For instance:

```
alert( 1 || 0 ); // 1 (1 is truthy)
alert( true || 'no matter what' ); // (true is truthy)

alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)
alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

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

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

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

We can use OR `||` for that:

```
let currentUser = null;
let defaultUser = "John";

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

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

If both `currentUser` and `defaultUser` were falsy then “`unnamed`” would be the result.

2. Short-circuit evaluation.

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

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

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

```
let x;

true || (x = 1);

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

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

```
let x;

false || (x = 1);
```

```
alert(x); // 1
```

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

As we can see, such a use case is a "shorter way to do `if`". The first operand is converted to boolean and if it's false then the second one is evaluated.

Most of time it's better to use a "regular" `if` to keep the code easy to understand, but sometimes that can be handy.

&& (AND)

The AND operator is represented with two ampersands `&&`:

```
result = a && b;
```

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

```
alert( true && true ); // true
alert( false && true ); // false
alert( true && false ); // false
alert( false && false ); // false
```

An example with `if`:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'Time is 12:30' );
}
```

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

```
if (1 && 0) { // evaluated as true && false
  alert( "won't work, because the result is falsy" );
}
```

AND seeks the first falsy value

Given multiple AND'ed values:

```
result = value1 && value2 && value3;
```

The AND `&&` operator does the following:

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

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

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

Examples:

```
// if the first operand is truthy,  
// AND returns the second operand:  
alert( 1 && 0 ); // 0  
alert( 1 && 5 ); // 5  
  
// if the first operand is falsy,  
// AND returns it. The second operand is ignored  
alert( null && 5 ); // null  
alert( 0 && "no matter what" ); // 0
```

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

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

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

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

i 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 `&&` were in parentheses: `(a && b) || (c && d)`.

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

For instance:

```
let x = 1;  
  
(x > 0) && alert( 'Greater than zero!' );
```

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

So we basically have an analogue for:

```
let x = 1;

if (x > 0) {
  alert( 'Greater than zero!' );
}
```

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

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

! (NOT)

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

The syntax is pretty simple:

```
result = !value;
```

The operator accepts a single argument and does the following:

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

For instance:

```
alert( !true ); // false
alert( !0 ); // true
```

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

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

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

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

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

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

Tasks

What's the result of OR?

importance: 5

What is the code below going to output?

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

[To solution](#)

What's the result of OR'ed alerts?

importance: 3

What will the code below output?

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

[To solution](#)

What is the result of AND?

importance: 5

What is this code going to show?

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

[To solution](#)

What is the result of AND'ed alerts?

importance: 3

What will this code show?

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

[To solution](#)

The result of OR AND OR

importance: 5

What will the result be?

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

[To solution](#)

Check the range between

importance: 3

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

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

[To solution](#)

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.

[To solution](#)

A question about “if”

importance: 5

Which of these `alert`s 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' );
```

[To solution](#)

Loops: while and for

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

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

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

The “while” loop

The `while` loop has the following syntax:

```
while (condition) {  
    // code  
    // so-called "loop body"  
}
```

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

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

```
let i = 0;  
while (i < 3) { // shows 0, then 1, then 2  
    alert( i );  
    i++;  
}
```

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

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

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

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

```
let i = 3;  
while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops  
    alert( i );  
    i--;  
}
```

Brackets are not required for a single-line body

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

```
let i = 3;  
while (i) alert(i--);
```

The “do...while” loop

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

```
do {  
    // loop body  
} while (condition);
```

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

For example:

```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

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

The “for” loop

The `for` loop is the most often used one.

It looks like this:

```
for (begin; condition; step) {  
    // ... loop body ...  
}
```

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

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

Let's examine the `for` statement part by part:

part

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

The general loop algorithm works like this:

```
Run begin
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ ...
```

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

Here's what exactly happens in our case:

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

// run begin
let i = 0
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// ...finish, because now i == 3
```

Inline variable declaration

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

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // error, no such variable
```

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

```
let i = 0;

for (i = 0; i < 3; i++) { // use an existing variable
  alert(i); // 0, 1, 2
}

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

Skipping parts

Any part of `for` can be skipped.

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

Like here:

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

for (; i < 3; i++) { // no need for "begin"
    alert( i ); // 0, 1, 2
}
```

We can also remove the `step` part:

```
let i = 0;

for (; i < 3;) {
    alert( i++ );
}
```

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

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

```
for (;;) {
    // repeats without limits
}
```

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

Breaking the loop

Normally the loop exits when the condition becomes falsy.

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

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

```
let sum = 0;

while (true) {

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

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

    sum += value;

}
alert( 'Sum: ' + sum );
```

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

```
alert .
```

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

Continue to the next iteration

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

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

The loop below uses `continue` to output only odd values:

```
for (let i = 0; i < 10; i++) {  
  
    // if true, skip the remaining part of the body  
    if (i % 2 == 0) continue;  
  
    alert(i); // 1, then 3, 5, 7, 9  
}
```

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

i The directive `continue` helps to decrease nesting level

A loop that shows odd values could look like this:

```
for (let i = 0; i < 10; i++) {  
  
    if (i % 2) {  
        alert( i );  
    }  
  
}
```

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

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

No break/continue to the right side of '?

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

For example, if we take this code:

```
if (i > 5) {  
    alert(i);  
} else {  
    continue;  
}
```

...And rewrite it using a question mark:

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

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

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

Labels for break/continue

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

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

```
for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Value at coords (${i},${j})` , '' );  
  
        // what if I want to exit from here to Done (below)?  
  
    }  
}  
  
alert('Done!');
```

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

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

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

```
labelName: for (...) {  
    ...
```

```
}
```

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

Like here:

```
outer: for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Value at coords (${i},${j})`);  
  
        // if an empty string or canceled, then break out of both loops  
        if (!input) break outer; // (*)  
  
        // do something with the value...  
    }  
}  
alert('Done!');
```

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

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

We can also move the label onto a separate line:

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

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

⚠ Labels are not a “goto”

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

For example, it is impossible to do this:

```
break label; // jumps to label? No.  
  
label: for (...)
```

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

Summary

We covered 3 types of loops:

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

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

If we don’t want to do anything on the current iteration and would like to forward to the next one, the `continue` directive does it.

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

✓ Tasks

Last loop value

importance: 3

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

```
let i = 3;

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

[To solution](#)

Which values does the while loop show?

importance: 4

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

Both loops `alert` the same values, or not?

1. The prefix form `++i`:

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

2. The postfix form `i++`

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

[To solution](#)

Which values get shown by the "for" loop?

importance: 4

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

Both loops `alert` same values or not?

1. The postfix form:

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

2. The prefix form:

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

[To solution](#)

Output even numbers in the loop

importance: 5

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

[Run the demo](#)

[To solution](#)

Replace "for" with "while"

importance: 5

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

```
for (let i = 0; i < 3; i++) {
  alert(`number ${i}!`);
```

[To solution](#)

Repeat until the input is correct

importance: 5

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

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

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

[Run the demo](#)

[To solution](#)

Output prime numbers

importance: 3

An 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.

[To solution](#)

The "switch" statement

A switch statement can replace multiple if checks.

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

The syntax

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

It looks like this:

```
switch(x) {  
    case 'value1': // if (x === 'value1')  
        ...  
        [break]  
  
    case 'value2': // if (x === 'value2')  
        ...  
        [break]  
  
    default:  
        ...  
        [break]  
}
```

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

An example

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

```
let a = 2 + 2;

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

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

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

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

An example without `break`:

```
let a = 2 + 2;

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

In the example above we'll see sequential execution of three `alert`s:

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

➊ 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 = 2 + 2;

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' )
}
```

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

Tasks

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!' );
}
```

[To solution](#)

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' );
}
```

[To solution](#)

Functions

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

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

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

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

Function Declaration

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

It looks like this:

```
function showMessage() {
    alert( 'Hello everyone!' );
}
```

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



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

For instance:

```
function showMessage() {
  alert( 'Hello everyone!' );
}

showMessage();
showMessage();
```

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

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

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

Local variables

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

For example:

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable

  alert( message );
}

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

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

Outer variables

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

```

let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John

```

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

For instance:

```

let userName = 'John';

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

  let message = 'Hello, ' + userName;
  alert(message);
}

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

showMessage();

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

```

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

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

```

let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

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

```

Global variables

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

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

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

Parameters

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

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

```
function showMessage(from, text) { // arguments: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

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

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

```
function showMessage(from, text) {

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

  alert( from + ': ' + text );
}

let from = "Ann";

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

// the value of "from" is the same, the function modified a local copy
alert( from ); // Ann
```

Default values

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

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

```
showMessage("Ann");
```

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

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

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

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

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

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

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() only executed if no text given
  // its result becomes the value of text
}
```

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()` is called everytime `someMessage()` is called without the `text` parameter. This is in contrast to some other languages like Python, where any default parameters are evaluated only once during the initial interpretation.

Default parameters old-style

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

For instance, an explicit check for being `undefined`:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'no text given';
  }

  alert( from + ": " + text );
}
```

...Or the `||` operator:

```
function showMessage(from, text) {
  // if text is falsy then text gets the "default" value
  text = text || 'no text given';
  ...
}
```

Returning a value

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

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

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

let result = sum(1, 2);
alert( result ); // 3
```

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

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

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}
```

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

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

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

For example:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

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

i A function with an empty `return` or without it returns `undefined`

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

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

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

An empty `return` is also the same as `return undefined`:

```
function doNothing() {
  return;
}

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

Never add a newline between `return` and the value

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

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

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

```
return;  
(some + long + expression + or + whatever * f(a) + f(b))
```

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

Naming a function

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

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

Ultrashort function names

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

For example, the [jQuery ↗](#) framework defines a function with `$`. The [LoDash ↗](#) library has its core function named `_`.

These are exceptions. Generally functions 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.

Summary

A function declaration looks like this:

```
function name(parameters, delimited, by, comma) {  
    /* code */  
}
```

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

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

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

Function naming:

- A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.
- A function is an action, so function names are usually verbal.

- There exist many well-known function prefixes like `create...`, `show...`, `get...`, `check...` and so on. Use them to hint what a function does.

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

✓ Tasks

Is "else" required?

importance: 4

The following function returns `true` if the parameter `age` is greater than `18`.

Otherwise it asks for a confirmation and returns its result:

```
function checkAge(age) {  
    if (age > 18) {  
        return true;  
    } else {  
        // ...  
        return confirm('Did parents allow you?');  
    }  
}
```

Will the function work differently if `else` is removed?

```
function checkAge(age) {  
    if (age > 18) {  
        return true;  
    }  
    // ...  
    return confirm('Did parents allow you?');  
}
```

Is there any difference in the behavior of these two variants?

[To solution](#)

Rewrite the function using '?' or '||'

importance: 4

The following function returns `true` if the parameter `age` is greater than `18`.

Otherwise it asks for a confirmation and returns its result.

```
function checkAge(age) {  
    if (age > 18) {  
        return true;  
    } else {
```

```
    return confirm('Do you have your parents permission to access this page?');
}
```

Rewrite it, to perform the same, but without `if`, in a single line.

Make two variants of `checkAge`:

1. Using a question mark operator `?`
2. Using OR `||`

[To solution](#)

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
```

[To solution](#)

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`.

[To solution](#)

Function expressions and arrows

In JavaScript, a function is not a “magical language structure”, but a special kind of value.

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

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

There is another syntax for creating a function that is called a *Function Expression*.

It looks like this:

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

Here, the function is created and assigned to the variable explicitly, like any other value. No matter how the function is defined, it's just a value stored in the variable `sayHi`.

The meaning of these code samples is the same: "create a function and put it into the variable `sayHi`".

We can even print out that value using `alert`:

```
function sayHi() {  
    alert( "Hello" );  
}  
  
alert( sayHi ); // shows the function code
```

Please note that the last line does not run the function, because there are no parentheses after `sayHi`. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

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

It is a special value of course, in the sense that we can call it like `sayHi()`.

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

We can copy a function to another variable:

```
function sayHi() {    // (1) create  
    alert( "Hello" );  
}  
  
let func = sayHi;    // (2) copy  
  
func(); // Hello    // (3) run the copy (it works)!  
sayHi(); // Hello   //      this still works too (why wouldn't it)
```

Here's what happens above in detail:

1. The Function Declaration (1) creates the function and puts it into the variable named sayHi.
2. Line (2) copies it into the variable func.

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

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

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

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

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

Everything would work the same. Even more obvious what's going on, right?

i Why is there a semicolon at the end?

You might wonder, why does Function Expression have a semicolon ; at the end, but Function Declaration does not:

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

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

The answer is simple:

- There's no need for ; at the end of code blocks and syntax structures that use them like if { ... }, for { }, function f { } etc.
- A Function Expression is used inside the statement: let sayHi = ... ;, as a value. It's not a code block. The semicolon ; is recommended at the end of statements, no matter what is the value. So the semicolon here is not related to the Function Expression itself in any way, it just terminates the statement.

Callback functions

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

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

question

Text of the question

yes

Function to run if the answer is “Yes”

no

Function to run if the answer is “No”

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

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

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

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

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

Before we explore how we can write it in a much shorter way, let’s note that in the browser (and on the server-side in some cases) such functions are quite popular. The major difference between a real-life implementation and the example above is that real-life functions use more complex ways to interact with the user than a simple `confirm`. In the browser, such a function usually draws a nice-looking question window. But that’s another story.

The arguments of `ask` are called *callback functions* or just *callbacks*.

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

We can use Function Expressions to write the same function much shorter:

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

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

Here, functions are declared right inside the `ask(...)` call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of `ask` (because they are not assigned to variables), but that's just what we want here.

Such code appears in our scripts very naturally, it's in the spirit of JavaScript.

i 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 see what is what in the code.

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

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

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

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

The more subtle difference is *when* a function is created by the JavaScript engine.

A Function Expression is created when the execution reaches it and is usable from then on.

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

Function Declarations are different.

A Function Declaration is usable in the whole script/code block.

In other words, when JavaScript *prepares* to run the script or a code block, it first looks for Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all of the Function Declarations are processed, the execution goes on.

As a result, a function declared as a Function Declaration can be called earlier than it is defined.

For example, this works:

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

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

The Function Declaration `sayHi` is created when JavaScript is preparing to start the script and is visible everywhere in it.

...If it was a Function Expression, then it wouldn't work:

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

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

Function Expressions are created when the execution reaches them. That would happen only in the line `(*)`. Too late.

When a Function Declaration is made within a code block, it is visible everywhere inside that block. But not outside of it.

Sometimes that's handy to declare a local function only needed in that block alone. But that feature may also cause problems.

For instance, let's imagine that we need to declare a function `welcome()` depending on the `age` variable that we get during runtime. And then we plan to use it some time later.

The code below doesn't work:

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

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

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

} else {

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

}

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

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

Here's another example:

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

if (age < 18) {
    welcome();           // \  (runs)
    // |
    function welcome() {
        // |
        alert("Hello!");
    }
}

// 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.

Now it works as intended:

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

let welcome;

if (age < 18) {

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

} else {

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

welcome(); // ok now
```

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

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

let welcome = (age < 18) ?
  function() { alert("Hello!"); } :
  function() { alert("Greetings!"); };

welcome(); // ok now
```

i When should you choose Function Declaration versus Function Expression?

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

It's also a little bit easier to look up `function f(...){...}` in the code than `let f = function(...){...}`. Function Declarations are more "eye-catching".

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

Arrow functions

There's one more very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions", because it looks like this:

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

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

In other words, it's roughly the same as:

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

...But much more concise.

Let's see an example:

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

/* The arrow function is a shorter form of:

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

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

If we have only one argument, then parentheses can be omitted, making that even shorter:

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

If there are no arguments, parentheses should be empty (but they should be present):

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

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

For instance, here's the rewritten example with `welcome()`:

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

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

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

Multiline arrow functions

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

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

Like this:

```
let sum = (a, b) => { // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, use return to get results
};

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

More to come

Here we praised arrow functions for brevity. But that's not all! Arrow functions have other interesting features. We'll return to them later in the chapter [Arrow functions revisited](#).

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

Summary

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

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

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

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

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

Tasks

Rewrite with arrow functions

Replace Function Expressions with arrow functions in the code:

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

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

[To solution](#)

JavaScript specials

This 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 mode

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

```
'use strict';  
  
...
```

The directive must be at the top of a script or at the beginning of a function.

Without `"use strict"`, everything still works, but some features behave in the old-fashion, “compatible” way. We’d generally prefer the modern behavior.

Some modern features of the language (like classes that we’ll study in the future) enable strict mode implicitly.

More in: [The modern mode, "use strict"](#).

Variables

Can be declared using:

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

A variable name can include:

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

Variables are dynamically typed. They can store any value:

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

There are 7 data types:

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

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

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

More in: [Variables](#) 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 pressed “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](#).

Operators

JavaScript supports the following operators:

Arithmetical

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

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

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

Assignments

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

Bitwise

Bitwise operators work with integers on bit-level: see the [docs ↗](#) when they are needed.

Ternary

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

Logical operators

Logical AND `&&` and OR `||` perform short-circuit evaluation and then return the value where it stopped.

Comparisons

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

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

Other comparisons convert to a number as well.

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

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

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

Logical operators

There are few others, like a comma operator.

More in: [Operators](#), [Comparisons](#), [Logical operators](#).

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” construct

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

For instance:

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

switch (age) {
  case 18:
    alert("Won't work"); // the result of prompt is a string, not a number

  case "18":
    alert("This works!");
    break;

  default:
    alert("Any value not equal to one above");
}
```

Details in: [The "switch" statement](#).

Functions

We covered three ways to create a function in JavaScript:

1. Function Declaration: the function in the main code flow

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

2. Function Expression: the function in the context of an expression

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

Function expressions can have a name, like `sum = function name(a, b)`, but that `name` is only visible inside that function.

3. Arrow functions:

```
// expression at 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. 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`.

Function Declaration	Function Expression
visible in the whole code block	created when the execution reaches it
-	can have a name, visible only inside the function

More: see [Functions, Function expressions and arrows](#).

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.

Code quality

This chapter explains coding practices that we'll use further in the development.

Debugging in Chrome

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

All modern browsers and most other environments support "debugging" – a special UI in developer tools that makes finding and fixing errors much easier.

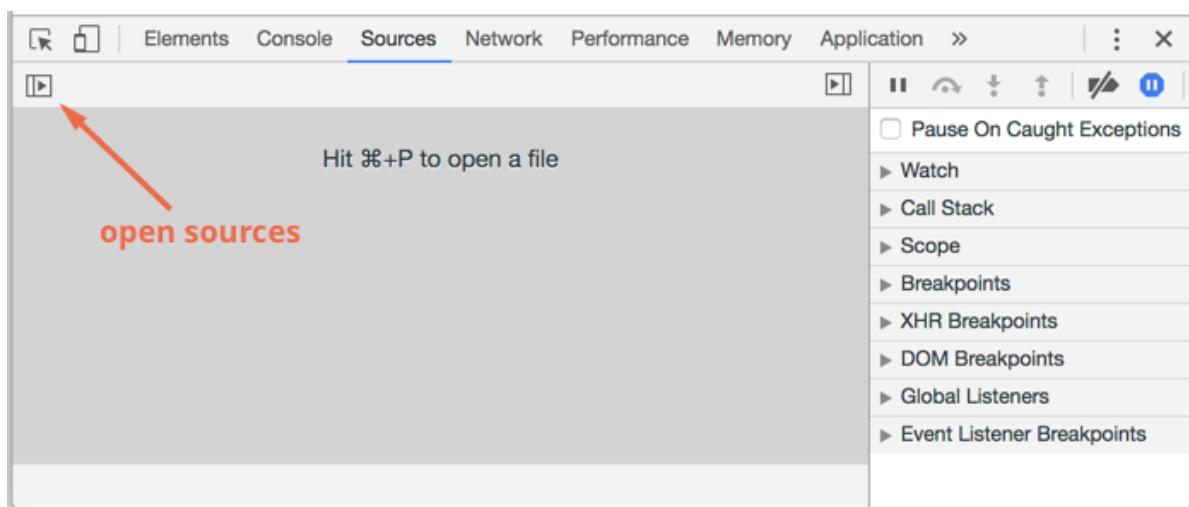
We'll be using Chrome here, because it's probably the most feature-rich in this aspect.

The “sources” pane

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

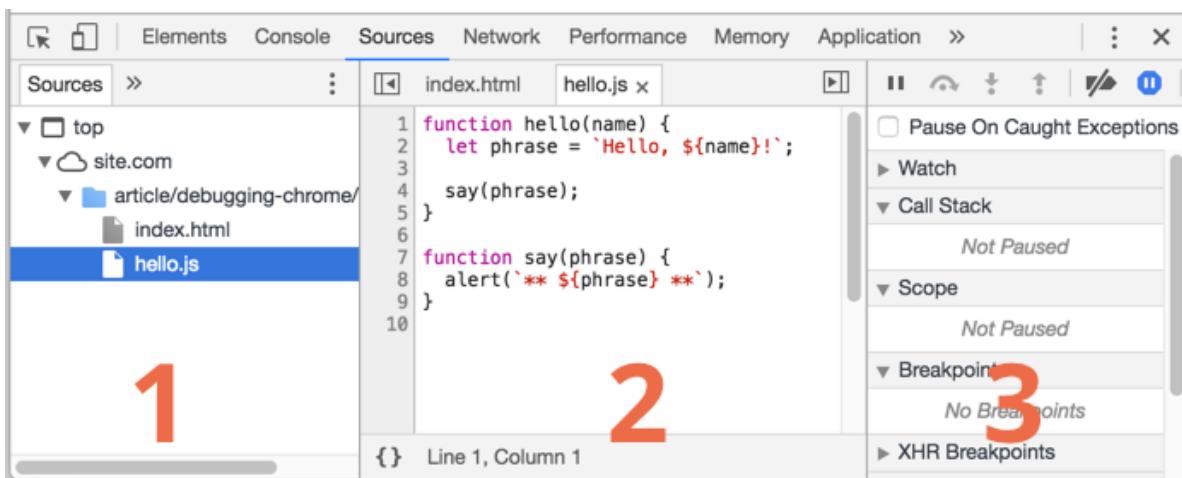
- Open the [example page](#) in Chrome.
- Turn on developer tools with `F12` (Mac: `Cmd+Opt+I`).
- Select the `Sources` pane.

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 `index.html` and then `hello.js` in the tree view. Here's what should show up:



Here we can see three zones:

1. The **Resources zone** lists HTML, JavaScript, CSS and other files, including images that are attached to the page. Chrome extensions may appear here too.
2. The **Source zone** shows the source code.
3. The **Information and control zone** 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`, and `hello("debugger")` returns nothing, so the result is `undefined`:

```

Console
Filter: top
All Errors Warnings Info Logs Debug Handled
> 1 + 2
<- 3
> hello("debugger")
<- undefined
>

```

Breakpoints

Let'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):

```
function hello(name) {  
    let phrase = `Hello, ${name}!`;  
    say(phrase);  
}  
function say(phrase) {  
    alert(`** ${phrase} **`);  
}
```

{ } Line 8, Column 3

Breakpoints

here's the list

Pause On Caught Exceptions
Watch
Call Stack
Scope
Breakpoints
hello.js:4
say(phrase);
hello.js:8
alert(`** \${phrase} **`)
XHR Breakpoints

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 pane. That's useful when we have many breakpoints in various files. It allows to:

- Quickly jump to the breakpoint in the code (by clicking on it in the right pane).
- 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 is truthy.

That's handy when we need to stop only for a certain variable value or for certain function parameters.

Debugger command

We can also pause the code by using the `debugger` command, like this:

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

That's very convenient when we are in a code editor and don't want to switch to the browser and look up the script in developer tools to set the breakpoint.

Pause and look around

In our example, `hello()` is called during the page load, so the easiest way to activate the debugger 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:

```
index.html | hello.js x
1 function hello(name) { name = "John"
2   let phrase = `Hello, ${name}!`; phrase = "Hello, John!"
3
4 say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
```

The right sidebar shows the debugger interface:

- Paused on breakpoint**: Shows the current state.
- Watch**: No Watch Expressions.
- Call Stack**:
 - hello (hello.js:4)
 - (anonymous) index.html:10
- Scope**:
 - Local**:
 - name: "John"
 - phrase: "Hello, John!"
 - this: Window
 - Global**: Window

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

1. **Watch** – shows current values for any expressions.

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

2. **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, the debugger jumps to the corresponding code, and all its variables can be examined as well.

3. **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 execution

Now it's time to *trace* the script.

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

▶ – 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 screenshot shows the Chrome DevTools Debugger interface. On the left, the code editor displays a file named 'hello.js' with the following content:1 function hello(name) { name = "John"
2 let phrase = 'Hello, \${name}!'; phrase = "Hello, John!"
3
4 say(phrase);
5 }
6
7 function say(phrase) { phrase = "Hello, John!"
8 alert(** \${phrase} **);
9 }
10

A red arrow points from the text "nested calls" to the line "say(phrase);". The status bar at the bottom indicates "Line 8, Column 3".

On the right, the developer tools sidebar shows the "Call Stack" expanded, revealing the current state of the call stack:

- say (hello.js:8)
- hello (hello.js:4)
- (anonymous) index.html:10

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.

⌚ – make a step (run the next command), but *don't go into the function*, hotkey **F10**.

If we click it now, `alert` will be shown. The important thing is that `alert` can be any function, the execution “steps over it”, skipping the function internals.

⚡ – make a step, hotkey **F11**.

The same as the previous one, but “steps into” nested functions. Clicking this will step through all script actions one by one.

↑ – continue the execution till the end of the current function, hotkey **Shift+F11**.

The execution would stop 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, and the developer tools is open, a script error automatically pauses the execution. Then we can analyze variables 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, but we’re too lazy to set a breakpoint.

To output something to console, 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 tab of developer tools or press `Esc` while in another tab: 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.

Summary

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

1. A breakpoint.
2. The `debugger` statements.
3. An error (if dev tools are open and the button  is "on")

Then we can examine variables and step on 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 as well!

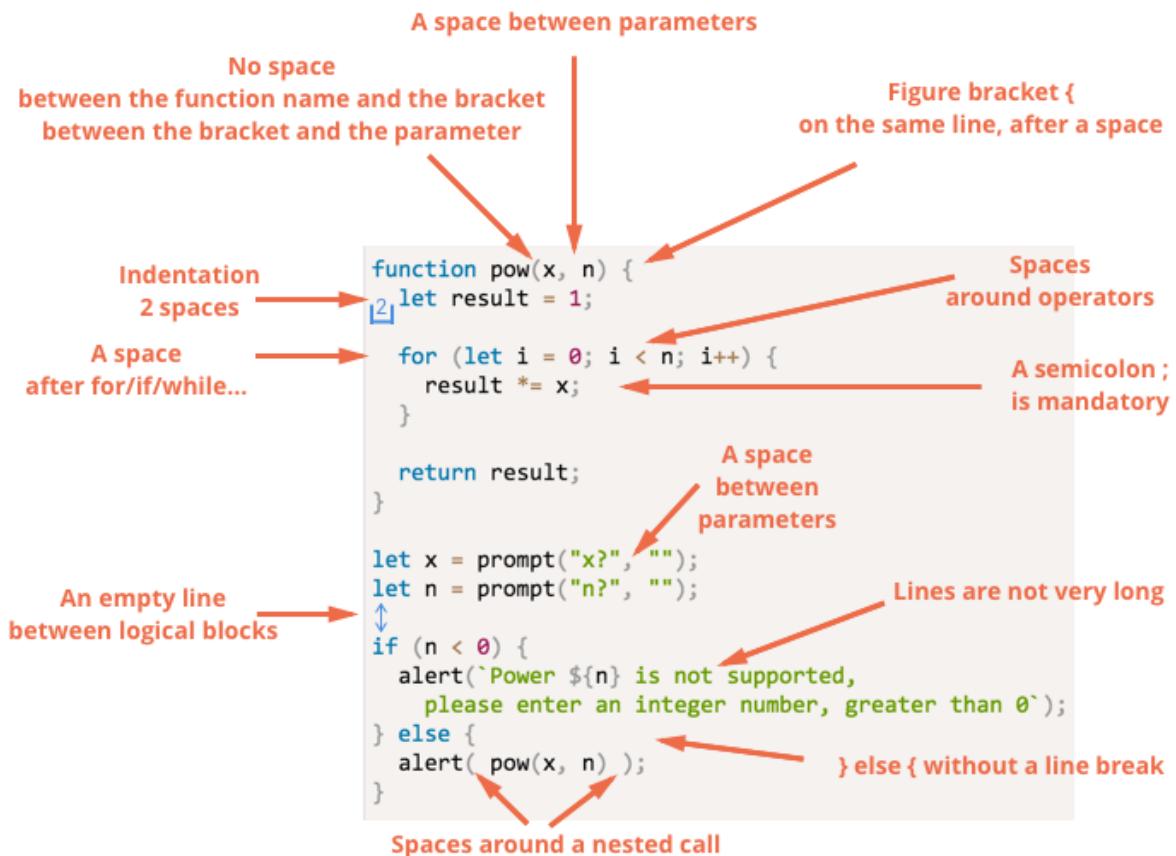
Coding Style

Our 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.

Syntax

Here is a cheatsheet with some suggested rules (see below for more details):



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

⚠ Irony Detected

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

Curly Braces

In 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 is an important edge case. Should we use brackets at all? If yes, then where?

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

 **Beginner sometimes do that. Bad!**
Figure brackets are not needed

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

 **Split to a separate line without brackets. Never do that!**
Source of nasty errors.

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

 **One line without brackets.**
Acceptable if it's short.

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

 **The best variant.**

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

In summary:

- For very short code, one line is acceptable. For example: `if (cond) return null.`
- But a separate line for each statement in brackets is usually easier to read.

Line Length

No one likes to read a long horizontal line of code. It's best practice to split them up and limit the length of your lines.

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 “Tab” symbol. Which one to choose is an old holy war. Spaces are more common nowadays.

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

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

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

- **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.

As you become more mature as a programmer, you may choose a no-semicolon style like [StandardJS ↗](#). Until then, it's best to use semicolons to avoid possible pitfalls.

Nesting Levels

Try to avoid nesting code too many levels deep.

Sometimes it's a good idea to use the “[continue](#)” directive in a loop 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 “edge 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.

1. Functions declared 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();
```

2. 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() {  
  ...  
}
```

3. 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 provides that information. Then, maybe we won't need to read the functions at all, especially if their names are descriptive of what they actually do.

Style Guides

A style guide contains general rules about “how to write” code, e.g. which quotes to use, how many spaces to indent, where to put line breaks, etc. A lot of minor things.

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

Of course, a team can always write their own style guide. Most of the time though, there's no need to. There are many existing tried and true options to choose from, so adopting one of these is usually your best bet.

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 cheatsheet at the beginning of this chapter. Once you've mastered that you can browse other style guides to pick up common principles and decide which one you like best.

Automated Linters

Linters are tools that can automatically check the style of your code and make suggestions for refactoring.

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, installing a linter is recommended even if you don't want to stick to one particular "code style".

Here are the most 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:

1. Install [Node.JS ↗](#).
2. Install ESLint with the command `npm install -g eslint` (`npm` is a JavaScript package installer).
3. Create a config file named `.eslintrc` in the root of your JavaScript project (in the folder that contains all your files).
4. 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,
    "semi": 1
  }
}
```

```
  },
  "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 <http://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, but all of them are debatable.

When we think about writing “better” code, the questions we should ask 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.

[To solution](#)

Comments

As 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.

From the first sight, commenting might be obvious, but novices in programming usually get it wrong.

Bad comments

Novices 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, 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 comments

So, 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 diagram language [UML](#) for high-level architecture diagrams. Definitely worth studying.

Document a function 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 <http://usejsdoc.org/>.

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:

1. You (or your colleague) open the code written some time ago, and see that it's "suboptimal".
2. You think: "How stupid I was then, and how much smarter I'm now", and rewrite using the "more obvious and correct" variant.

3. ...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 only if it’s impossible to make the code so simple and self-descriptive that it doesn’t require those.

Comments are also used for auto-documenting tools like JSDoc3: they read them and generate HTML-docs (or docs in another format).

Ninja code

Learning without thought is labor lost; thought without learning is perilous.

“ Confucius

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 wit

Make 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 variables

The Dao hides in wordlessness. Only the Dao is well begun and well completed.

“ Laozi (Tao Te Ching)

Another way to code faster 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 abbreviations

If 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, he'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 hardest thing of all is to find a black cat in a dark room, especially if there is no cat.



Confucius

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 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.



Laozi (Tao Te Ching)

There are already enough names.

One must know when to stop.

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. 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 he's 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 `niceItem` 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.

Overlap outer variables

When in the light, can't see anything in the darkness.

“ Guan Yin Zi

When in the darkness, can see everything in the light.

Use same names for variables inside and outside a function. As simple. No efforts required.

```
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 he'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.

Summary

All “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.

Automated testing with mocha

Automated testing will be used in further tasks.

It's actually a part of the “educational minimum” of a developer.

Why 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 can be executed easily and check all the main use cases.

Behavior Driven Development (BDD)

Let's use a technique named [Behavior Driven Development ↗](#) or, in short, BDD. That approach is used among many projects. BDD is not just about testing. That's more.

BDD is three things in one: tests AND documentation AND examples.

Enough words. Let's see the example.

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 looks 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. Uses to group "workers" – the `it` blocks. In our case we're describing the function `pow`.

it("title", 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 see further.

The development flow

The flow of development usually looks like this:

1. An initial spec is written, with tests for the most basic functionality.
2. An initial implementation is created.
3. To check whether it works, we run the testing framework [Mocha ↗](#) (more details soon) that runs the spec. Errors are displayed. We make corrections until everything works.
4. Now we have a working initial implementation with tests.
5. We add more use cases to the spec, probably not yet supported by the implementations. Tests start to fail.
6. Go to 3, update the implementation till tests give no errors.
7. 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.

In our case, the first step is complete: we have an initial spec for `pow`. So let's make an implementation. But before that let's make a "zero" run of the spec, just to see that tests are working (they will all fail).

The spec in action

Here 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>

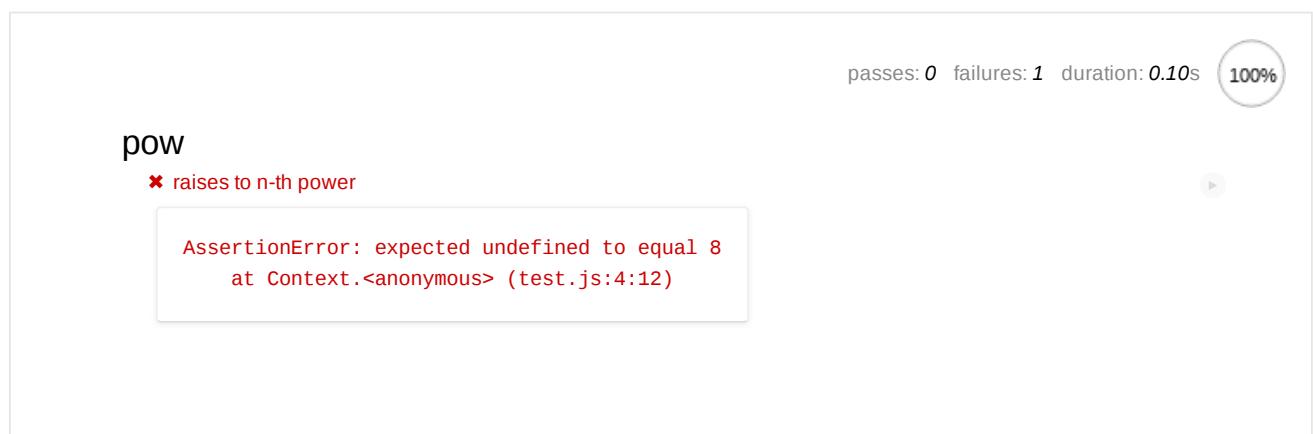
</html>

```

The page can be divided into five parts:

1. The `<head>` – add third-party libraries and styles for tests.
2. The `<script>` with the function to test, in our case – with the code for `pow`.
3. The tests – in our case an external script `test.js` that has `describe("pow", ...)` from above.
4. The HTML element `<div id="mocha">` will be used by Mocha to output results.
5. 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 advanced test-runners, like [karma ↗](#) and others. So it's generally not a problem to setup many different tests.

Initial implementation

Let's make a simple implementation of `pow`, for tests to pass:

```

function pow() {
  return 8; // :) we cheat!
}

```

Wow, now it works!

pow

✓ raises to n-th power



Improving the spec

What 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 see if `pow(3, 4) = 81`.

We can select one of two ways to organize the test here:

1. 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);  
  });  
  
});
```

2. 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 3 is 27", function() {  
    assert.equal(pow(3, 3), 27);  
  });  
  
});
```

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:

A screenshot of a test runner interface. At the top right, it shows 'passes: 1' and 'failures: 1' with a duration of '1.46s' and a 100% coverage circle. Below this, the test name 'pow' is shown. Underneath, there are two test cases: a green checkmark next to '2 raised to power 3 is 8' and a red X next to '3 raised to power 3 is 27'. A red box highlights the error message: 'AssertionError: expected 8 to equal 27' followed by 'at Context.<anonymous> (test.js:8:12)'. There are also two small circular arrows on the right side of the test results.

As we could expect, the second test failed. Sure, our function always returns `8`, while the `assert` expects `27`.

Improving the implementation

Let'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);
    });
  }
})
```

```

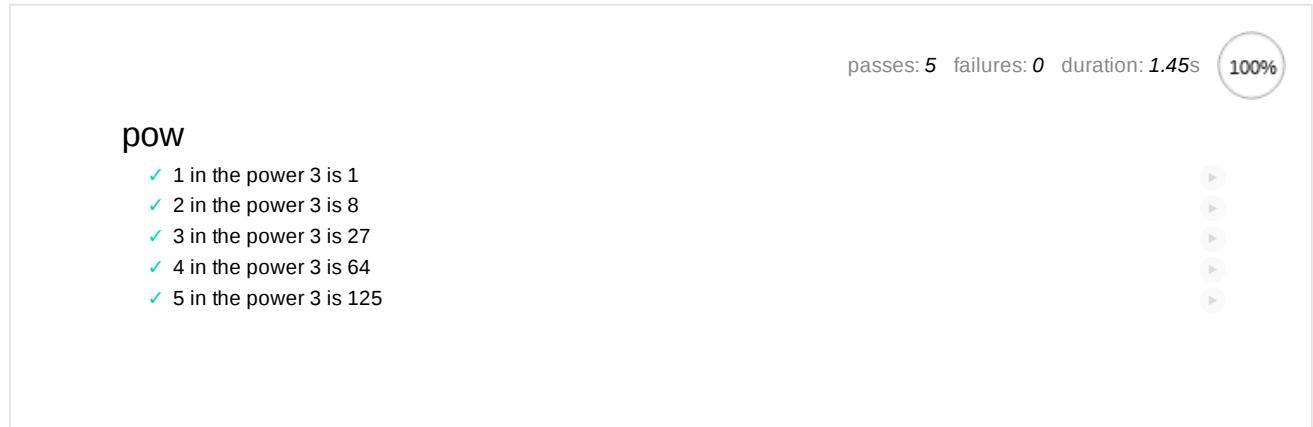
    }

    for (let x = 1; x <= 5; x++) {
      makeTest(x);
    }

  });
}

```

The result:



```

  passes: 5  failures: 0  duration: 1.45s  100%

```

pow

- ✓ 1 in the power 3 is 1
- ✓ 2 in the power 3 is 8
- ✓ 3 in the power 3 is 27
- ✓ 4 in the power 3 is 64
- ✓ 5 in the power 3 is 125

Nested describe

We'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 n", 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:

pow

raises x to power n

- ✓ 1 in the power 3 is 1
- ✓ 2 in the power 3 is 8
- ✓ 3 in the power 3 is 27
- ✓ 4 in the power 3 is 64
- ✓ 5 in the power 3 is 125



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`.

i 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` (`before/each`) are used to perform initialization, zero out counters or do something else between the tests (or test groups).

Extending the spec

The 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 screenshot shows a test runner interface with the following details:

- Top right: passes: 5, failures: 2, duration: 1.43s, 100% coverage.
- Test suite: **pow**
- Test 1: ✖ if n is negative, the result is NaN
AssertionError: expected 1 to be NaN
at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:23:12)
at Context.<anonymous> (test.js:19:12)
- Test 2: ✖ if n is not integer, the result is NaN
AssertionError: expected 4 to be NaN
at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:23:12)
at Context.<anonymous> (test.js:23:12)
- Test 3: raises x to power n
✓ 1 in the power 3 is 1
✓ 2 in the power 3 is 8
✓ 3 in the power 3 is 27
✓ 4 in the power 3 is 64
✓ 5 in the power 3 is 125

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.

i Other assertions

Please note the assertion `assert.isNaN`: it checks for `Nan`.

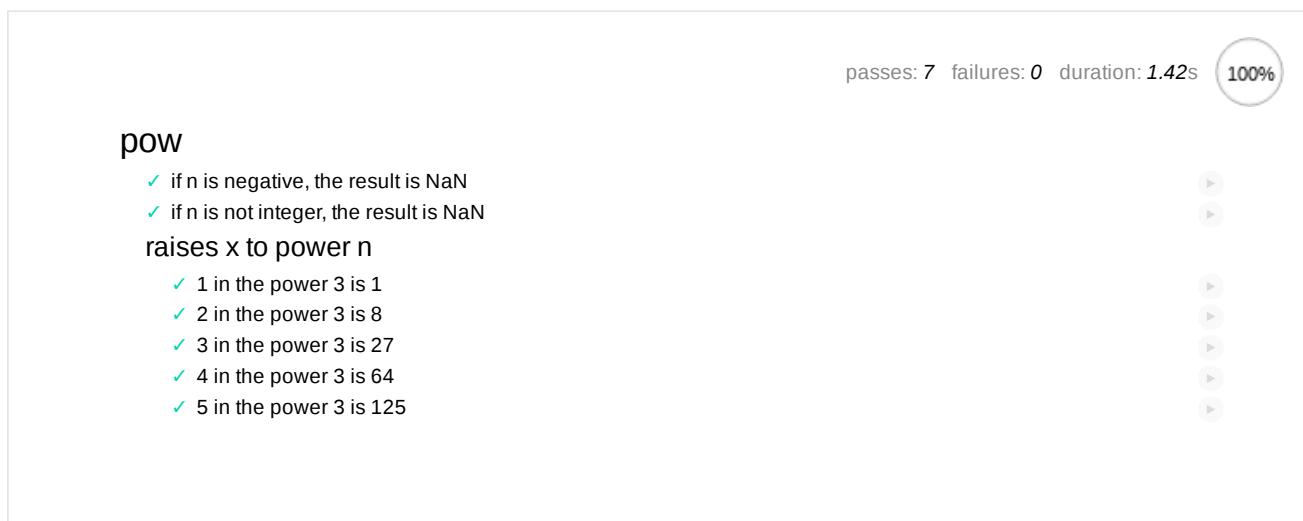
There are other assertions in Chai as well, for instance:

- `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:



passes: 7 failures: 0 duration: 1.42s 100%

pow

- ✓ if n is negative, the result is NaN
- ✓ if n is not integer, the result is NaN
- raises x to power n
 - ✓ 1 in the power 3 is 1
 - ✓ 2 in the power 3 is 8
 - ✓ 3 in the power 3 is 27
 - ✓ 4 in the power 3 is 64
 - ✓ 5 in the power 3 is 125

[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:

1. **Tests** guarantee that the code works correctly.
2. **Docs** – the titles of `describe` and `it` tell what the function does.
3. **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:

1. To perform the change, no matter what. And then our users meet bugs and report them. If we can afford that.
2. Or people become afraid to modify such functions, if the punishment for errors is harsh. Then it becomes old, overgrown with cobwebs, no one wants to get into it, and that's not good.

Automatically tested code is contrary to that!

If the project is covered with tests, there's just no such problem. 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 it's easier to change and improve it. But not only that.

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.

What now?

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.

Tasks

What's wrong in the test?

importance: 5

What'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.

[To solution](#)

Polyfills

The 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 the 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).

Babel

When we use modern features of the language, some engines may fail to support such code. Just as said, not all features are implemented everywhere.

Here Babel comes to the rescue.

Babel is a [transpiler](#). It rewrites modern JavaScript code into the previous standard.

Actually, there are two parts in Babel:

1. First, the transpiler program, which rewrites the code. The developer runs it on their own computer. It rewrites the code into the older standard. And then the code is delivered to the website for users. Modern project build system like [webpack](#) or [brunch](#) provide means to run transpiler automatically on every code change, so that doesn't involve any time loss from our side.

2. Second, the polyfill.

The transpiler rewrites the code, so syntax features are covered. But for new functions we need to write a special script that implements them. JavaScript is a highly dynamic language, scripts may not just add new functions, but also modify built-in ones, so that they behave according to the modern standard.

There's a term "polyfill" for scripts that "fill in" the gap and add missing implementations.

Two interesting polyfills are:

- [babel polyfill ↗](#) that supports a lot, but is big.
- [polyfill.io ↗](#) service that allows to load/construct polyfills on-demand, depending on the features we need.

So, we need to setup the transpiler and add the polyfill for old engines to support modern features.

If we orient towards modern engines and do not use features except those supported everywhere, then we don't need to use Babel.

Examples in the tutorial

As you're reading the offline version, examples are not runnable. But they usually work :)

[Chrome Canary ↗](#) is good for all examples, but other modern browsers are mostly fine too.

Note that on production we can use Babel to translate the code into suitable for less recent browsers, so there will be no such limitation, the code will run everywhere.

Objects: the basics

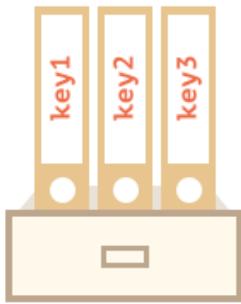
Objects

As we know from the chapter [Data types](#), there are seven data types in JavaScript. Six 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:

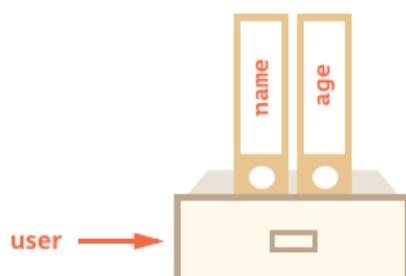
```
let user = {
    name: "John", // by key "name" store value "John"
    age: 30        // by key "age" store value 30
};
```

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:

1. The first property has the name `"name"` and the value `"John"`.
2. 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".



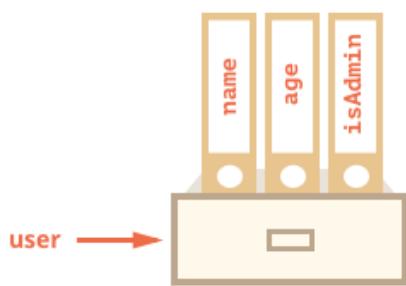
We can add, remove and read files from it any time.

Property values are accessible using the dot notation:

```
// get fields of the object:  
alert( user.name ); // John  
alert( user.age ); // 30
```

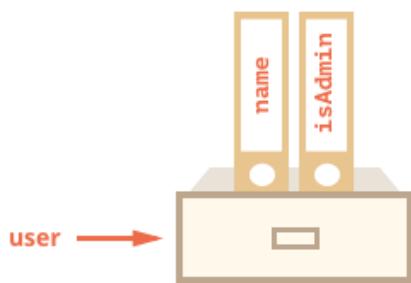
The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```



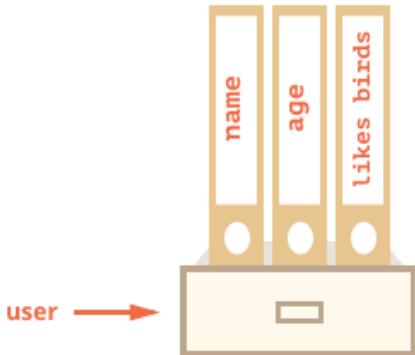
To remove a property, we can use `delete` operator:

```
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
```

That's because the dot requires the key to be a valid variable identifier. That is: no spaces and other limitations.

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. The dot notation cannot be used in a similar way.

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")
```

Computed properties

We can use square brackets in an object literal. 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 the 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.

Reserved words are allowed as property names

A variable cannot have a name equal to one of language-reserved words like “for”, “let”, “return” etc.

But for an object property, there's no such restriction. Any name is fine:

```
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return ); // 6
```

Basically, any name is allowed, but there's a special one: `"__proto__"` that gets special treatment for historical reasons. For instance, we can't set it to a non-object value:

```
let obj = {};
obj.__proto__ = 5;
alert(obj.__proto__); // [object Object], didn't work as intended
```

As we see from the code, the assignment to a primitive `5` is ignored.

That can become a source of bugs and even vulnerabilities if we intend to store arbitrary key-value pairs in an object, and allow a visitor to specify the keys.

In that case the visitor may choose “`proto`” as the key, and the assignment logic will be ruined (as shown above).

There is a way to make objects treat `__proto__` as a regular property, which we'll cover later, but first we need to know more about objects. There's also another data structure [Map](#), that we'll learn in the chapter [Map, Set, WeakMap and WeakSet](#), which supports arbitrary keys.

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
};
```

Existence check

A notable objects feature is that it's possible to access any property. There will be no error if the property doesn't exist! Accessing a non-existing property just returns `undefined`. It provides a very common way to test whether the property exists – to get it and compare vs `undefined`:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

There also exists a special operator `"in"` to check for the existence of a property.

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 would mean a variable containing the actual name will be tested. For instance:

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, takes the name from key and checks for such property
```

i Using “in” for properties that store `undefined`

Usually, the strict comparison `"== undefined"` check 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` is usually not 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 object

Are 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 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:

```
// 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" => not integer property
alert( String(Math.trunc(Number("1.2")))); // "1", not same "1.2" => 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.

Copying by reference

One of the fundamental differences of objects vs primitives is that they are stored and copied “by reference”.

Primitive values: strings, numbers, booleans – are assigned/copied “as a whole value”.

For instance:

```
let message = "Hello!";
let phrase = message;
```

As a result we have two independent variables, each one is storing the string "Hello!" .

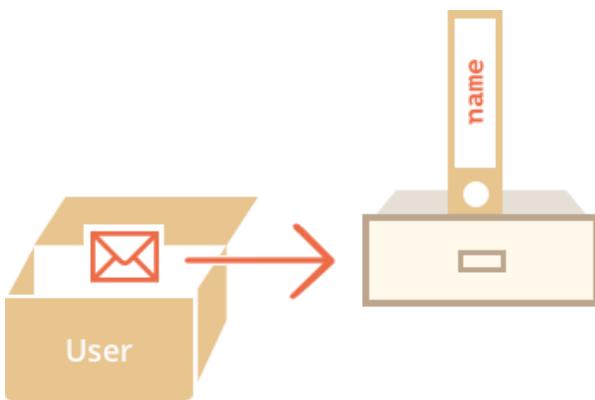


Objects are not like that.

A variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.

Here's the picture for the object:

```
let user = {
  name: "John"
};
```



Here, the object is stored somewhere in memory. And the variable `user` has a “reference” to it.

When an object variable is copied – the reference is copied, the object is not duplicated.

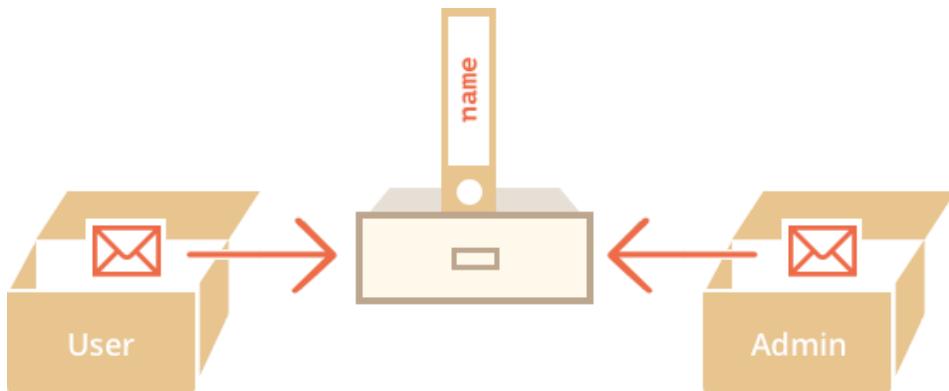
If we imagine an object as a cabinet, then a variable is a key to it. Copying a variable duplicates the key, but not the cabinet itself.

For instance:

```
let user = { name: "John" };

let admin = user; // copy the reference
```

Now we have two variables, each one with the reference to the same object:



We can use any variable to access the cabinet 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
```

The example above demonstrates that there is only one object. As if we had a cabinet with two keys and used one of them (`admin`) to get into it. Then, if we later use the other key (`user`) we would see changes.

Comparison by reference

The equality `==` and strict equality `===` operators for objects work exactly the same.

Two objects are equal only if they are the same object.

For instance, two variables reference the same object, 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 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 necessary very rarely and usually are a result of a coding mistake.

Const object

An object declared as `const` can be changed.

For instance:

```
const user = {  
    name: "John"  
};  
  
user.age = 25; // (*)  
  
alert(user.age); // 25
```

It might seem that the line `(*)` would cause an error, but no, there's totally no problem. That's because `const` fixes the value of `user` itself. And here `user` stores the reference to the same object all the time. The line `(*)` goes *inside* the object, it doesn't reassign `user`.

The `const` would give an error if we try to set `user` to something else, for instance:

```
const user = {  
    name: "John"  
};  
  
// Error (can't reassign user)  
user = {  
    name: "Pete"  
};
```

...But what if we want to make constant object properties? So that `user.age = 25` would give an error. That's possible too. We'll cover it in the chapter [Property flags and descriptors](#).

Cloning and merging, Object.assign

So, copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object? Create an independent copy, a clone?

That's also doable, but a little bit more difficult, because there's no built-in method for that in JavaScript. Actually, that's rarely needed. Copying by reference is good most of the time.

But if we really want that, then we need to 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 clone
clone.name = "Pete"; // changed the data in it

alert( user.name ); // still John in the original object

```

Also we can use the method [Object.assign ↗](#) for that.

The syntax is:

```
Object.assign(dest[, src1, src2, src3...])
```

- Arguments `dest`, and `src1, ..., srcN` (can be as many as needed) are objects.
- It copies the properties of all objects `src1, ..., srcN` into `dest`. In other words, properties of all arguments starting from the 2nd are copied into the 1st. Then it returns `dest`.

For instance, we can use it to merge several objects into one:

```

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 }

```

If the receiving object (`user`) already has the same named property, it will be overwritten:

```

let user = { name: "John" };

// overwrite name, add isAdmin
Object.assign(user, { name: "Pete", isAdmin: true });

// now user = { name: "Pete", isAdmin: true }

```

We also can use `Object.assign` to replace the loop for simple cloning:

```

let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);

```

It copies all properties of `user` into the empty object and returns it. Actually, the same as the loop, but shorter.

Until now we assumed that all properties of `user` are primitive. But properties can be references to other objects. What to do with them?

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 the `user.sizes` is an object, it will be copied by reference. So `clone` and `user` will share the same `sizes`:

Like this:

```

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++;           // change a property from one place
alert(clone.sizes.width); // 51, see the result from the other one

```

To fix that, we should use the 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”.

There's a standard algorithm for deep cloning that handles the case above and more complex cases, called the [Structured cloning algorithm ↗](#). In order not to reinvent the wheel, we can use

a working implementation of it from the JavaScript library [lodash](#), the method is called `_.cloneDeep(obj)`.

Summary

Objects 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 to take 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`.

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. 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` or [_.cloneDeep\(obj\)](#).

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.

Tasks

Hello, object

importance: 5

Write the code, one line for each action:

1. Create an empty object `user`.
2. Add the property `name` with the value `John`.
3. Add the property `surname` with the value `Smith`.
4. Change the value of the `name` to `Pete`.
5. Remove the property `name` from the object.

[To solution](#)

Check for emptiness

importance: 5

Write 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.](#) ↗

[To solution](#)

Constant objects?

importance: 5

Is it possible to change an object declared with `const`? What do you think?

```
const user = {  
  name: "John"  
};  
  
// does it work?  
user.name = "Pete";
```

[To solution](#)

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`.

[To solution](#)

Multiply numeric properties by 2

importance: 3

Create a function `multiplyNumeric(obj)` that multiplies all numeric properties 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.](#) ↗

[To solution](#)

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.

1. There's a base set of inherently reachable values, that cannot be deleted for obvious reasons.

For instance:

- Local variables and parameters of the current function.
- Variables and parameters for other functions on the current chain of nested calls.
- Global variables.
- (there are some other, internal ones as well)

These values are called *roots*.

2. Any other value is considered reachable if it's reachable from a root by a reference or by a chain of references.

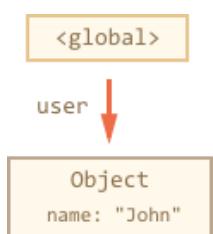
For instance, if there's an object in a local variable, and that object has a property referencing another object, that object is considered reachable. And those that it references are also reachable. Detailed examples to follow.

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;
```



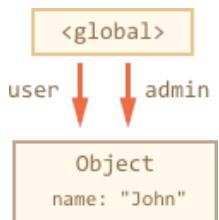
Now John becomes unreachable. There's no way to access it, no references to it. Garbage collector will junk the data and free the memory.

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's in memory. If we overwrite `admin` too, then it can be removed.

Interlinked objects

Now a more complex example. The family:

```

function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

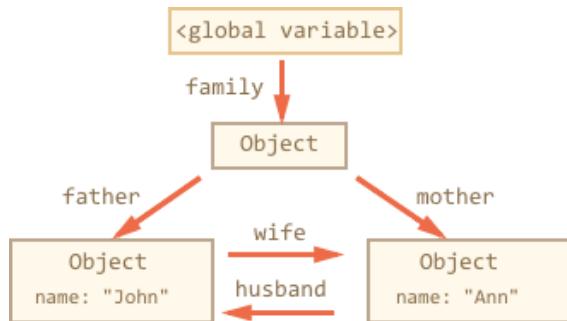
  return {
    father: man,
    mother: woman
  }
}

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:



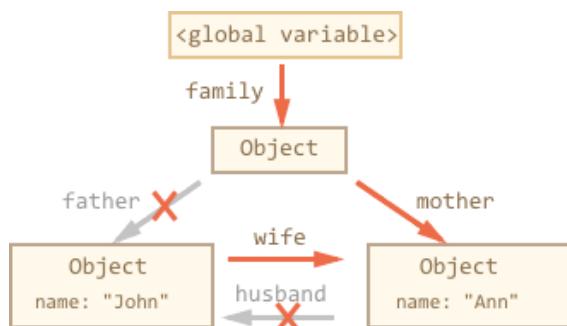
As of now, all objects are reachable.

Now let's remove two references:

```

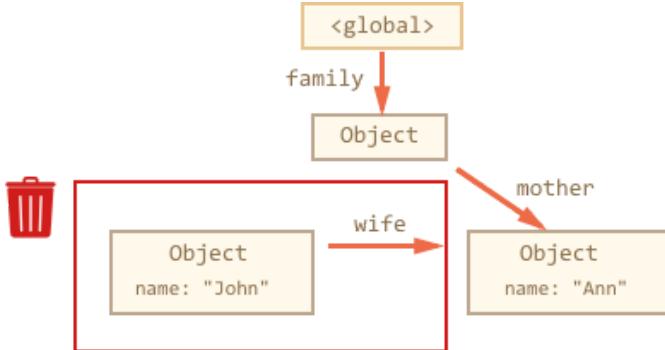
delete family.father;
delete family.mother.husband;

```



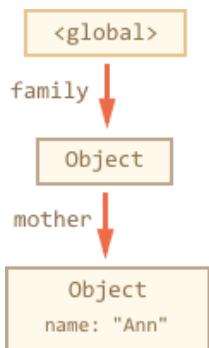
It's not enough to delete only one of these two references, because all objects would still be reachable.

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 unaccessible.

After garbage collection:



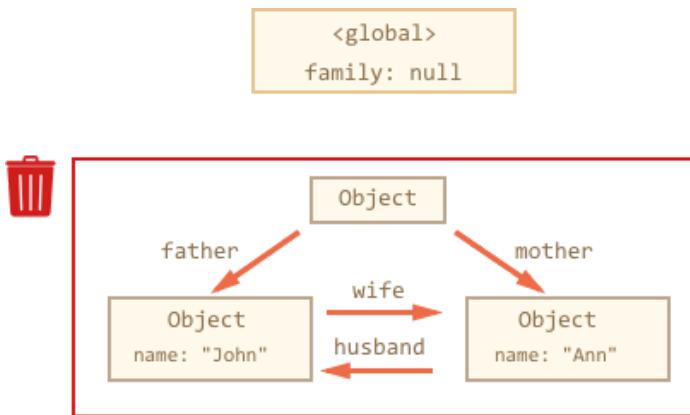
Unreachable island

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.

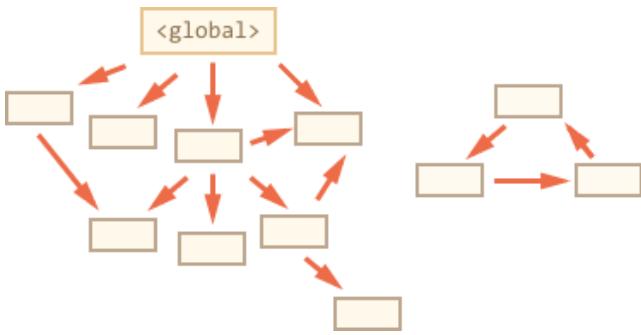
Internal algorithms

The basic garbage collection algorithm is called "mark-and-sweep".

The following "garbage collection" steps are regularly performed:

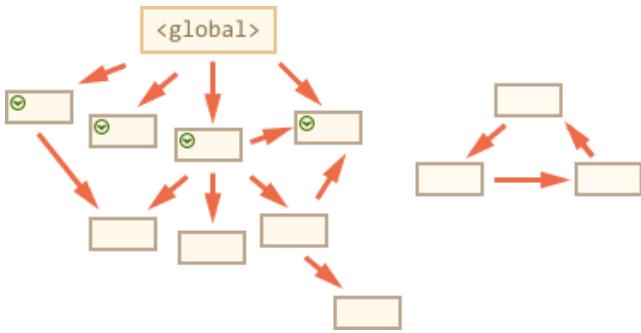
- The garbage collector takes roots and "marks" (remembers) them.
- Then it visits and "marks" all references from them.
- Then it visits marked objects and marks *their* references. All visited objects are remembered, so as not to visit the same object twice in the future.
- ...And so on until there are unvisited references (reachable from the roots).
- All objects except marked ones are removed.

For instance, let our object structure look like this:

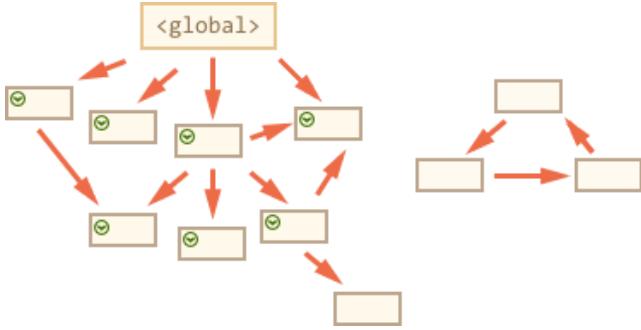


We can clearly see an "unreachable island" to the right side. Now let's see how "mark-and-sweep" garbage collector deals with it.

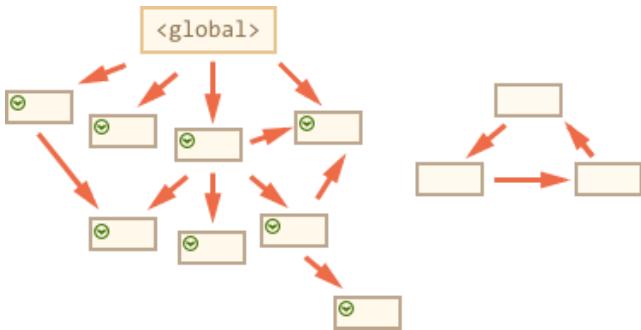
The first step marks the roots:



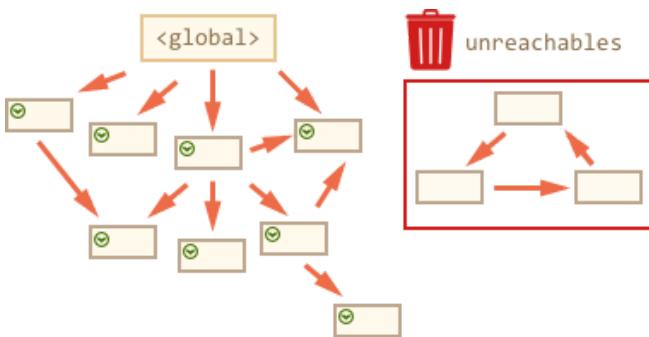
Then their references are marked:



...And their references, while possible:



Now the objects that could not be visited in the process are considered unreachable and will be removed:



That's the concept of how garbage collection works.

JavaScript engines apply many optimizations to make it run faster and not affect the execution.

Some of the optimizations:

- **Generational collection** – objects are split into two sets: “new ones” and “old ones”. Many objects appear, do their job and die fast, they can be cleaned up aggressively. 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 tries to split the garbage collection into pieces. Then the pieces are executed one by one, separately. That requires some extra bookkeeping between them to track changes, but we have 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 are 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 going 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.

Summary

The 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.

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, the more detailed information about V8 garbage collector is in the article [A tour of V8: Garbage Collection ↗](#).

[V8 blog ↗](#) also publishes articles about changes in memory management from time to time.

Naturally, to learn the 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 V8 engineers. I'm saying: “V8”, because it is best covered with articles in 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.

Symbol type

By specification, object property keys may be either of string type, or of symbol type. Not numbers, not booleans, only strings or symbols, these two types.

Till now we've only seen strings. Now let's see the advantages that symbols can give us.

Symbols

“Symbol” value represents a unique identifier.

A value of this type can be created using `Symbol()`:

```
// id is a new symbol
let id = Symbol();
```

We can also give symbol 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 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.

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
```

If we really want to show a symbol, we need to call `.toString()` on it, like here:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), now it works
```

That's a “language guard” against messing up, because strings and symbols are fundamentally different and should not occasionally convert one into another.

“Hidden” properties

Symbols allow us to create “hidden” properties of an object, that no other part of code can occasionally access or overwrite.

For instance, if we want to store an “identifier” for the object `user`, we can use a symbol as a key for it:

```
let user = { name: "John" };
let id = Symbol("id");

user[id] = "ID Value";
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"`?

Let's make the example a bit deeper to see that.

Imagine that another script wants to have its own "id" property inside `user`, for its own purposes. That may be another JavaScript library, so the scripts are completely unaware of each other.

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, because symbols are always different, even if they have the same name.

Now note that 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 = "ID Value";

// ...if later another script the uses "id" for its purposes...

user.id = "Their id value"
// boom! overwritten! it did not mean to harm the colleague, but did it!
```

Symbols in a literal

If we want to use a symbol in an object literal, we need square brackets.

Like this:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not just "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...in`

Symbolic 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] );

```

That's a part of the general “hiding” concept. 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`).

Property keys of other types are coerced to strings

We can only use strings or symbols as keys in objects. Other types are 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)

```

Global symbols

As we've seen, usually all symbols are different, even if they have the same names. 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 create or read a symbol in 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
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 right for global symbols.

Symbol.keyFor

For global symbols, not only `Symbol.for(key)` returns a symbol by name, but there's a reverse call: `Symbol.keyFor(sym)`, that does the reverse: returns a name by a global symbol.

For instance:

```
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// get name from 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 return `undefined`.

For instance:

```
alert( Symbol.keyFor(Symbol.for("name")) ); // name, global symbol

alert( Symbol.keyFor(Symbol("name2")) ); // undefined, the argument isn't a global symbol
```

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.

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` return exactly the same symbol.

Symbols have two main use cases:

1. “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 occasionally listed. Also it won't be accessed directly, because another script does not have our symbol, so it will not occasionally intervene into its actions.

So we can “covertly” hide something into objects that we need, but others should not see, using symbolic properties.

2. 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. So they are not really hidden. But most libraries, built-in methods and syntax constructs

adhere to a common agreement that they are. And the one who explicitly calls the aforementioned methods probably understands well what he's doing.

Object 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 examples

For the 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 the function and assign it to the property `user.sayHi` of the object.

Then we can call it. The user can now speak!

A function that is the 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 an [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. We'll scratch the surface of that topic later in the chapter [Objects, classes, inheritance](#).

Method shorthand

There exists a shorter syntax for methods in an object literal:

```
// these objects do the same  
  
let user = {  
    sayHi: function() {  
        alert("Hello");  
    }  
};  
  
// method shorthand looks better, right?  
let 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() {
    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"
  }
};
```

...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(); // Whoops! inside sayHi(), the old name is used! error!
```

If we used `this.name` instead of `user.name` inside the `alert`, then the code would work.

“this” is not bound

In JavaScript, “this” keyword behaves unlike most other programming languages. First, it can be used in any function.

There's no syntax error in the code like that:

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

The value of `this` is evaluated during the run-time. And it can be anything.

For instance, the same function may have different “this” when called from different objects:

```
let user = { name: "John" };
let admin = { name: "Admin" };

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

// use the same functions 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)
```

Actually, we can 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 (if one forgets `use strict`) the value of `this` in such case will be the *global object* (`window` in a browser, we'll get to it later). This is a historical behavior that “`use strict`” fixes.

Please note that usually a call of a function that uses `this` without an object is not normal, but rather a programming mistake. If a function has `this`, then it is usually meant to be called in the context of an object.

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's the object "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, greater flexibility opens a place 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 evade problems.

Internals: Reference Type

In-depth language feature

This section covers an advanced topic, to understand certain edge-cases better.

If you want to go on faster, it can be skipped or postponed.

An intricate method call can lose `this`, for instance:

```
let user = {
  name: "John",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
};

user.hi(); // John (the simple call 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 ternary operator that chooses either `user.hi` or `user.bye`. In this case the result is `user.hi`.

The method is immediately called with parentheses `()`. But it doesn't work right!

You can see that the call results in an error, cause 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.

Looking closely, we may notice two operations in `obj.method()` statement:

1. First, the dot `'.'` retrieves the property `obj.method`.
2. 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.
- `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).

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). Later in this tutorial, we will learn various ways to solve this problem such as `func.bind()`.

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.
- That 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.

Tasks

Syntax check

importance: 2

What is the result of this code?

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.S. There's a pitfall :)

[To solution](#)

Explain the value of "this"

importance: 3

In the code below we intend to call `user.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
```

[To solution](#)

Using "this" in object literal

importance: 5

Here 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?
```

[To solution](#)

Create a calculator

importance: 5

Create an object `calculator` with three methods:

- `read()` prompts for two values and saves them as object properties.
- `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.](#) ↗

[To solution](#)

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
```

Modify the code of `up` and `down` to make the calls chainable, like this:

```
ladder.up().up().down().showStep(); // 1
```

Such approach is widely used across JavaScript libraries.

[Open a sandbox with tests.](#) ↗

[To solution](#)

Object to primitive conversion

What happens when objects are added `obj1 + obj2`, subtracted `obj1 - obj2` or printed using `alert(obj)`?

There are special methods in objects that do the conversion.

In 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 close it.

For objects, there's no to-boolean conversion, because all objects are `true` in a boolean context. So there are only string and numeric 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 like `alert(obj)` and in similar contexts.

ToPrimitive

When an object is used in the context where a primitive is required, for instance, in an `alert` or mathematical operations, it's converted to a primitive value using the `ToPrimitive` algorithm ([specification](#) ↗).

That algorithm allows us to customize the conversion using a special object method.

Depending on the context, the conversion has a so-called "hint".

There are three variants:

`"string"`

When an operation expects a string, for object-to-string conversions, like `alert`:

```
// output
alert(obj);

// using object as a property key
anotherObj[obj] = 123;
```

"number"

When an operation expects a number, for object-to-number conversions, like 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;
```

"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 both strings and numbers would do. Or when an object is compared using `==` with a string, number or a symbol.

```
// binary plus
let total = car1 + car2;

// obj == string/number/symbol
if (user == 1) { ... };
```

The greater/less operator `<>` can work with both strings and numbers too. Still, it uses “number” hint, not “default”. That’s for historical reasons.

In practice, all built-in objects except for one case (`Date` object, we’ll learn it later) implement “default” conversion the same way as “number”. And probably we should do the same.

Please note – there are only three hints. It’s that simple. There is no “boolean” hint (all objects are `true` in boolean context) or anything else. And if we treat “default” and “number” the same, like most built-ins do, then there are only two conversions.

To do the conversion, JavaScript tries to find and call three object methods:

1. Call `obj[Symbol.toPrimitive](hint)` if the method exists,
2. Otherwise if hint is “string”
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is “number” or “default”

- try `obj.valueOf()` and `obj.toString()`, whatever exists.

Symbol.toPrimitive

Let'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) {
  // return a primitive value
  // hint = one of "string", "number", "default"
}
```

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

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.

If there's no `Symbol.toPrimitive` then JavaScript tries to find them and try in the order:

- `toString -> valueOf` for “string” hint.
- `valueOf -> toString` otherwise.

For instance, here `user` does the same as above using a combination of `toString` and `valueOf`:

```
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

```

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.

ToPrimitive and ToString/ToNumber

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 a hint “number”.

The only mandatory thing: these methods must return a primitive.

An operation that initiated the conversion gets that primitive, and then continues to work with it, applying further conversions if necessary.

For instance:

- Mathematical operations (except binary plus) perform `ToNumber` conversion:

```

let obj = {
  toString() { // toString handles all conversions in the absence of other methods
    return "2";
  }
};

alert(obj * 2); // 4, ToPrimitive gives "2", then it becomes 2

```

- Binary plus checks the primitive – if it's a string, then it does concatenation, otherwise it performs `ToNumber` and works with numbers.

String example:

```

let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 (ToPrimitive returned string => concatenation)

```

Number example:

```

let obj = {
  toString() {
    return true;
  }
};

alert(obj + 2); // 3 (ToPrimitive returned boolean, not string => ToNumber)

```

Historical notes

For historical reasons, methods `toString` or `valueOf` *should* return a primitive: if any of them returns an object, then there's no error, but that object is ignored (like if the method didn't exist).

In contrast, `Symbol.toPrimitive` *must* return a primitive, otherwise, there will be an error.

Summary

The 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 string conversions)
- `"number"` (for maths)

- "default" (few operators)

The specification describes explicitly which operator uses which hint. There are very few operators that “don’t know what to expect” and use the "default" hint. Usually for built-in objects "default" hint is handled the same way as "number", so in practice the last two are often merged together.

The conversion algorithm is:

1. Call `obj[Symbol.toPrimitive](hint)` if the method exists,
2. Otherwise if hint is "string"
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is "number" or "default"
 - try `obj.valueOf()` and `obj.toString()`, whatever exists.

In practice, it's often enough to implement only `obj.toString()` as a “catch-all” method for all conversions that return a “human-readable” representation of an object, for logging or debugging purposes.

Constructor, operator "new"

The regular `{...}` syntax allows 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 function

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. 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 as `new User(...)`, it does the following steps:

1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.

3. 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 the result of `new User("Jack")` is the same object 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 can be used as a constructor. That is: any function 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 constructor function, like this:

```
let user = new function() {
  this.name = "John";
  this.isAdmin = false;

  // ...other code for user creation
  // maybe complex logic and statements
  // local variables etc
};
```

The 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.

Dual-syntax constructors: 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 empty 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 to allow both `new` and regular calls to work the same. That is, create the same object:

```
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 object, then it 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 an 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; // finishes the execution, 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`, if it has no arguments:

```
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 constructor

Using 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() { ... }
}
*/
```

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, in the chapter [Objects, classes, inheritance](#) we return to objects and cover them in-depth, including inheritance and classes.

Tasks

Two functions – one object

importance: 2

Is it possible to create functions `A` and `B` such as `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.

[To solution](#)

Create new Calculator

importance: 5

Create a constructor function `Calculator` that creates objects with 3 methods:

- `read()` asks for two values using `prompt` and remembers them in object properties.
- `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 demo](#)

[Open a sandbox with tests.](#) ↗

[To solution](#)

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.](#) ↗

[To solution](#)

Data types

More data structures and more in-depth study of the types.

Methods of primitives

JavaScript 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 6 primitive types: `string`, `number`, `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. But as properties and methods are very useful in programming, JavaScript engines try to optimize them to reduce the additional burden.

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 as methods.
- Primitives must be as fast and lightweight as possible.

The solution looks a little bit awkward, but here it is:

1. Primitives are still primitive. A single value, as desired.
2. The language allows access to methods and properties of strings, numbers, booleans and symbols.
3. When this happens, a special “object wrapper” is created that provides the extra functionality, and then is destroyed.

The “object wrappers” are different for each primitive type and are called: `String`, `Number`, `Boolean` and `Symbol`. Thus, they provide different sets of methods.

For instance, there exists a method `str.toUpperCase()` ↗ that returns a capitalized string.

Here's how it works:

```
let str = "Hello";
alert( str.toUpperCase() ); // HELLO
```

Simple, right? Here's what actually happens in `str.toUpperCase()`:

1. 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()`.
2. That method runs and returns a new string (shown by `alert`).
3. 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 create “wrapper objects” for primitives explicitly 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 1 ); // "number"  
  
alert( typeof new Number(1) ); // "object"!
```

And because what follows, `zero`, is an object, 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 a totally sane 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.

✓ Tasks

Can I add a string property?

importance: 5

Consider the following code:

```
let str = "Hello";  
  
str.test = 5;  
  
alert(str.test);
```

How do you think, will it work? What will be shown?

[To solution](#)

Numbers

All numbers in JavaScript are stored in 64-bit format [IEEE-754](#), also known as “double precision”.

Let’s recap and expand upon what we currently know about them.

More ways to write a number

Imagine we need to write 1 billion. The obvious way is:

```
let billion = 1000000000;
```

But in real life we usually avoid writing a long string of zeroes as it’s easy to mistype. Also, we are lazy. We will usually 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 shorten a number by appending the letter `e` to the number and specifying the zeroes count:

```
let billion = 1e9; // 1 billion, literally: 1 and 9 zeroes
```

```
alert( 7.3e9 ); // 7.3 billions (7,300,000,000)
```

In other words, "e" multiplies the number by 1 with the given zeroes count.

```
1e3 = 1 * 1000  
1.23e6 = 1.23 * 1000000
```

Now let's write something very small. Say, 1 microsecond (one millionth of a second):

```
let ms = 0.000001;
```

Just like before, using "e" can help. If we'd like to avoid writing the zeroes explicitly, we could say:

```
let ms = 1e-6; // six 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)
```

Hex, binary and octal numbers

Hexadecimal ↗ 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)`.

Rounding

One 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 and -1.1 becomes -1.

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:

1. 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.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. 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 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:
`+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 (they are zero for integer numbers), and 1 bit is for the sign.

If a number is too big, it would overflow the 64-bit storage, potentially giving an infinity:

```
alert( 1e500 ); // Infinity
```

What may be a little less obvious, but happens quite often, is the loss of precision.

Consider this (falsy!) 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! There are more consequences than an incorrect comparison here. Imagine you're making an e-shopping site and the visitor puts `$0.10` and `$0.20` goods into their chart. 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 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.

In other words, 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 `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", so the number shows up as 0.3. But beware, the loss still exists.

We can see this in action:

```
alert( 0.1.toFixed(20) ); // 0.1000000000000000555
```

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, there're a number of ways:

1. We can 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;
+sum.toFixed(2); // 0.3
```

2. We can temporarily turn numbers into integers for the maths and then revert it back. It works like this:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
```

This works because when we do 0.1 * 10 = 1 and 0.2 * 10 = 2 then both numbers become integers, and there's no precision loss.

3. If we were dealing with a shop, then the most radical solution would be to store all prices in cents and use no fractions at all. But what if we apply a discount of 30%? In practice, totally evading fractions is rarely feasible, so the solutions above help avoid this pitfall.

i 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.

i 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 every number can be positive or negative, 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`? Sorry, but the answer is no. 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`.

Compare with `Object.is`

There is a special built-in method `Object.is` ↗ that compares values like `==`, but is more reliable for two edge cases:

1. It works with `Nan`: `Object.is(NaN, NaN) === true`, that's a good thing.
2. Values `0` and `-0` are different: `Object.is(0, -0) === false`, it rarely matters, but these values technically are different.

In all other cases, `Object.is(a, b)` is the same as `a == b`.

This way of comparison is 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
```

i 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...) / Math.min(a, b, c...)

Returns the greatest/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 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 big numbers:

- Append `"e"` with the zeroes count to the number. Like: `123e6` is `123` with 6 zeroes.
- A negative number after `"e"` causes the number to be divided by 1 with given zeroes. That's for one-millionth or such.

For different numeral systems:

- Can write numbers directly in hex (`0x`), octal (`0o`) and binary (`0b`) systems
- `parseInt(str, base)` parses an integer from any numeral system with base: `2 ≤ base ≤ 36`.
- `num.toString(base)` converts a number to a string in the numeral system with the given `base`.

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.

Tasks

Sum numbers from the visitor

importance: 5

Create a script that prompts the visitor to enter two numbers and then shows their sum.

[Run the demo](#)

P.S. There is a gotcha with types.

[To solution](#)

Why `6.35.toFixed(1) == 6.3?`

importance: 4

According 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?

[To solution](#)

Repeat until the input is a number

importance: 5

Create 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 demo](#)

[Open a sandbox with tests.](#) ↗

[To solution](#)

An occasional infinite loop

importance: 4

This loop is infinite. It never ends. Why?

```
let i = 0;
while (i != 10) {
    i += 0.2;
}
```

[To solution](#)

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
```

[To solution](#)

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( random(1, 5) ); // 1
alert( random(1, 5) ); // 3
alert( random(1, 5) ); // 5
```

You can use the solution of the [previous task](#) as the base.

[To solution](#)

Strings

In 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.

Quotes

Let'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, including function calls:

```
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
```

If we try to use single or double quotes in the same way, there will 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. You can read more about it in the [docs ↗](#). This

is called “tagged templates”. This feature makes it easier to wrap strings into custom templating or other functionality, but it is rarely used.

Special characters

It is still possible to create multiline strings with single 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
```

For example, these two lines describe the same:

```
alert( "Hello\nWorld" ); // two lines using a "newline symbol"  
  
// two lines using a normal newline and backticks  
alert( `Hello  
World` );
```

There are other, less common “special” characters as well. Here’s the list:

Character	Description
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\uNNNN</code>	A unicode symbol with the hex code <code>NNNN</code> , for instance <code>\u00A9</code> – is a unicode for the copyright symbol © . It must be exactly 4 hex digits.
<code>\u{NNNNNNNN}</code>	Some rare characters are encoded with two unicode symbols, taking up to 4 bytes. This long unicode requires braces around it.

Examples with unicode:

```
alert( "\u00A9" ); // ©  
alert( "\u{20331}" ); // 俗, a rare chinese hieroglyph (long unicode)  
alert( "\u{1F60D}" ); // 😊, a smiling face symbol (another long unicode)
```

All special characters start with a backslash character `\`. It is also called an “escape character”.

We would also use it if we want to insert a quote into the 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, that refers only to the quotes that are same as the enclosing ones. So, as a more elegant solution, we could switch to double quotes or backticks instead:

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

Note that the backslash `\` serves for the correct reading of the string by JavaScript, then disappears. The in-memory string has no `\`. You can clearly see that in `alert` from the examples above.

But what if we need to show an actual backslash `\` within the string?

That's possible, but we need to double it like `\\"\\`:

```
alert( `The backslash: \\\\" ); // The backslash: \
```

String length

The `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.

Accessing characters

To get a character at position `pos`, use square brackets `[pos]` or call the method `str.charAt(pos)` ↗. The first character starts from the zero position:

```
let str = `Hello`;  
  
// the first character  
alert( str[0] ); // H  
alert( str.charAt(0) ); // H
```

```
// the last character
alert( str[str.length - 1] ); // o
```

The square brackets are a modern way of getting a character, while `charAt` exists mostly for historical reasons.

The only difference between them is that if no character is found, `[]` returns `undefined`, and `charAt` returns an empty string:

```
let str = `Hello`;
alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (an empty string)
```

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 immutable

Strings 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 case

Methods `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 search starting from the 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 laid 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);
}
```

i `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 `if` 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!
}
```

The bitwise NOT trick

One of the old tricks used here is the **bitwise NOT**  `~` operator. It converts the number to a 32-bit integer (removes the decimal part if exists) and then reverses all bits in its binary representation.

For 32-bit integers the call `~n` means exactly the same as `-(n+1)` (due to IEEE-754 format).

For instance:

```
alert( ~2 ); // -3, the same as -(2+1)
alert( ~1 ); // -2, the same as -(1+1)
alert( ~0 ); // -1, the same as -(0+1)
alert( ~-1 ); // 0, the same as -(-1+1)
```

As we can see, `~n` is zero only if `n == -1`.

So, the test `if (~str.indexOf("..."))` is truthy that the result of `indexOf` is not `-1`. In other words, when there is a match.

People use it to shorten `indexOf` checks:

```
let str = "Widget";

if (~str.indexOf("Widget")) {
  alert( 'Found it!' ); // works
}
```

It is usually not recommended to use language features in a non-obvious way, but this particular trick is widely used in old code, so we should understand it.

Just remember: `if (~str.indexOf(...))` reads as “if found”.

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( "Midget".includes("id") ); // true
alert( "Midget".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.substring(start [, end])`

Returns the part of the string *between* `start` and `end`.

This is almost the same as `slice`, but it allows `start` to be greater than `end`.

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
```

Let's recap these methods to avoid any confusion:

method	selects...	negatives
<code>slice(start, end)</code>	from <code>start</code> to <code>end</code> (not including <code>end</code>)	allows negatives
<code>substring(start, end)</code>	between <code>start</code> and <code>end</code>	negative values mean <code>0</code>
<code>substr(start, length)</code>	from <code>start</code> get <code>length</code> characters	allows negative <code>start</code>

i 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.

The author finds themselves using `slice` almost all the time.

Comparing strings

As we know from the chapter [Comparisons](#), strings are compared character-by-character in alphabetical order.

Although, there are some oddities.

1. A lowercase letter is always greater than the uppercase:

```
alert( 'a' > 'Z' ); // true
```

2. 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, let's review the internal representation of strings in JavaScript.

All strings 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 the code for the character at position `pos`:

```
// different case letters have different codes
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

String.fromCodePoint (code)

Creates a character by its numeric code

```
alert( String.fromCodePoint(90) ); // Z
```

We can also add unicode characters by their codes using \u followed by the hex code:

```
// 90 is 5a in hexadecimal system  
alert( '\u005a' ); // Z
```

Now let's see the characters with codes 65 .. 220 (the latin alphabet and a little bit extra) by making a string of them:

See? Capital characters go first, then a few special ones, then lowercase characters.

Now it becomes obvious why $a \geq z$

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 Ö stand apart from the main alphabet. Here, its code is greater than anything from a to z.

Correct comparisons

The “right” algorithm to do string comparisons is more complex than it may seem, because alphabets are different for different languages. The same-looking letter may be located differently in different alphabets.

So, the browser needs to know the language to compare.

Luckily, all modern browsers (IE10+ requires the additional library [Intl.JS](#)) 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 1 if str is greater than str2 according to the language rules.
- Returns -1 if str is less than str2.
- Returns 0 if they are equal.

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) and setup additional rules like case sensitivity or should "a" and "á" be treated as the same etc.

Internals, Unicode

Advanced knowledge

The section goes deeper into string internals. This knowledge will be useful for you if you plan to deal with emoji, rare mathematical or hieroglyphs characters or other rare symbols.

You can skip the section if you don't plan to support them.

Surrogate pairs

Most symbols have a 2-byte code. Letters in most European languages, numbers, and even most hieroglyphs, have a 2-byte representation.

But 2 bytes only allow 65536 combinations and that's not enough for every possible symbol. So rare symbols are encoded with a pair of 2-byte characters called “a surrogate pair”.

The length of such symbols is 2:

```
alert( 'Ӯ'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL X
alert( 'Ӧ'.length ); // 2, FACE WITH TEARS OF JOY
```

```
alert( '𩫱'.length ); // 2, a rare chinese hieroglyph
```

Note that surrogate pairs did not exist at the time when JavaScript was created, and thus are not correctly processed by the language!

We actually have a single symbol in each of the strings above, but the `length` shows a length of 2.

`String.fromCodePoint` and `str.codePointAt` are few rare methods that deal with surrogate pairs right. They recently appeared in the language. Before them, there were only `String.fromCharCode ↗` and `str.charCodeAt ↗`. These methods are actually the same as `fromCodePoint/codePointAt`, but don't work with surrogate pairs.

But, for instance, getting a symbol can be tricky, because surrogate pairs are treated as two characters:

```
alert( '𩫱'[0] ); // strange symbols...
alert( '𩫱'[1] ); // ...pieces of the surrogate pair
```

Note that pieces of the surrogate pair have no meaning without each other. So the alerts in the example above actually display garbage.

Technically, surrogate pairs are also detectable by their codes: if a character has the code in the interval of `0xd800..0xdbff`, then it is the first part of the surrogate pair. The next character (second part) must have the code in interval `0xdc00..0xdfff`. These intervals are reserved exclusively for surrogate pairs by the standard.

In the case above:

```
// charCodeAt is not surrogate-pair aware, so it gives codes for parts
alert( '𩫱'.charCodeAt(0).toString(16) ); // d835, between 0xd800 and 0xdbff
alert( '𩫱'.charCodeAt(1).toString(16) ); // dc3, between 0xdc00 and 0xdfff
```

You will find more ways to deal with surrogate pairs later in the chapter [Iterables](#). There are probably special libraries for that too, but nothing famous enough to suggest here.

Diacritical marks and normalization

In many languages there are symbols that are composed of the base character with a mark above/under it.

For instance, the letter `a` can be the base character for: `àáâãäåã`. Most common “composite” character have their own code in the UTF-16 table. But not all of them, because there are too many possible combinations.

To support arbitrary compositions, UTF-16 allows us to use several unicode characters. The base character and one or many “mark” characters that “decorate” it.

For instance, if we have `S` followed by the special “dot above” character (code `\u0307`), it is shown as `Ś`.

```
alert( 'S\u0307' ); // Š
```

If we need an additional mark above the letter (or below it) – no problem, just add the necessary mark character.

For instance, if we append a character “dot below” (code `\u0323`), then we’ll have “S with dots above and below”: `฿`.

For example:

```
alert( 'S\u0307\u0323' ); // Š
```

This provides great flexibility, but also an interesting problem: two characters may visually look the same, but be represented with different unicode compositions.

For instance:

```
alert( 'S\u0307\u0323' ); // Š, S + dot above + dot below
alert( 'S\u0323\u0307' ); // Š, S + dot below + dot above

alert( 'S\u0307\u0323' == 'S\u0323\u0307' ); // false
```

To solve this, there exists a “unicode normalization” algorithm that brings each string to the single “normal” form.

It is implemented by `str.normalize()` ↗.

```
alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() ); // true
```

It’s funny that in our situation `normalize()` actually brings together a sequence of 3 characters to one: `\u1e68` (S with two dots).

```
alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

In reality, this is not always the case. The reason being that the symbol `฿` is “common enough”, so UTF-16 creators included it in the main table and gave it the code.

If you want to learn more about normalization rules and variants – they are described in the appendix of the Unicode standard: [Unicode Normalization Forms](#) ↗, but for most practical purposes the information from this section is enough.

Summary

- There are 3 types of quotes. Backticks allow a string to span multiple lines and embed expressions.
- Strings in JavaScript are encoded using UTF-16.
- We can use special characters like `\n` and insert letters by their unicode using `\u.....`.
- To get a character, use: `[]`.
- To get a substring, use: `slice` or `substring`.
- To lowercase/uppercase a string, use: `toLowerCase/toUpperCase`.
- To look for a substring, use: `indexOf`, or `includesstartsWith/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. See the [manual ↗](#) for details.

Strings also have methods for doing search/replace with regular expressions. But that topic deserves a separate chapter, so we’ll return to that later.

✓ Tasks

Uppercast 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. ↗](#)

[To solution](#)

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 xxxx') == true
checkSpam("innocent rabbit") == false
```

[Open a sandbox with tests.](#)

[To solution](#)

Truncate the text

importance: 5

Create a function `truncate(str, maxlen)` that checks the length of the `str` and, if it exceeds `maxlen` – 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.](#)

[To solution](#)

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.](#)

[To solution](#)

Arrays

Objects 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.

Declaration

There 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.

Methods pop/push, shift/unshift

A [queue ↗](#) is one of 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 meet it very often. For example, a queue of messages that need to be shown on-screen.

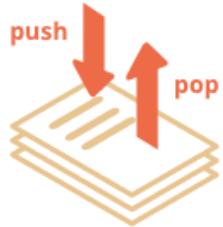
There's another use case for arrays – the data structure named [stack ↗](#).

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 it 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
```

`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 );
```

Internals

An array is a special kind of object. The square brackets used to access a property `arr[0]` actually come from the object syntax. 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 7 basic types in JavaScript. 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[99999] = 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.

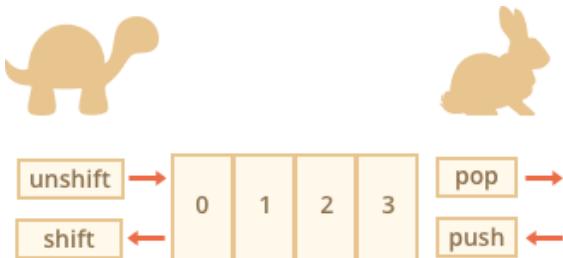
The ways to misuse an array:

- 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 `{}`.

Performance

Methods `push/pop` run fast, while `shift/unshift` are slow.



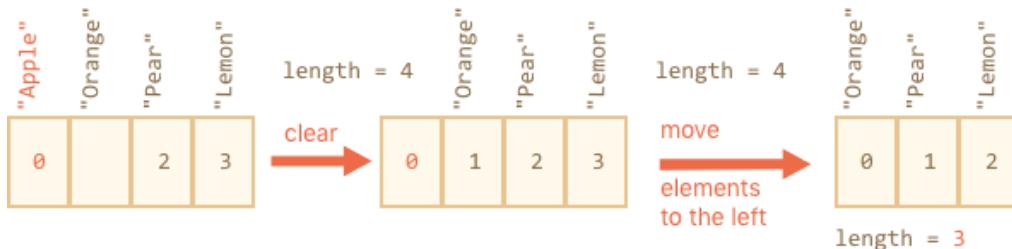
Why is it faster to work with the end of an array than with its beginning? Let's see what happens during the execution:

```
fruits.shift(); // take 1 element from the start
```

It's not enough to take and remove the element with the number `0`. Other elements need to be renumbered as well.

The `shift` operation must do 3 things:

1. Remove the element with the index `0`.
2. Move all elements to the left, renumber them from the index `1` to `0`, from `2` to `1` and so on.
3. Update the `length` property.



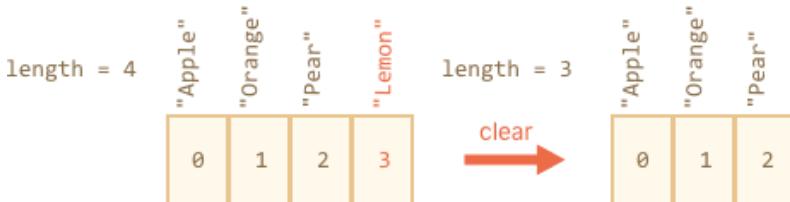
The more elements in the array, the more time to move them, more in-memory operations.

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:

1. 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.

2. 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 matter only in bottlenecks or just irrelevant. 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 themself 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
```

In the code above, `new Array(number)` has all elements `undefined`.

To evade 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, to store matrices:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

```
alert( matrix[1][1] ); // the central element
```

toString

Arrays 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"
```

Summary

Array 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.

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.

We will return to arrays and study more methods to add, remove, extract elements and sort arrays in the chapter [Array methods](#).

✓ Tasks

Is array copied?

importance: 3

What 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 ); // ?
```

[To solution](#)

Array operations.

importance: 5

Let's try 5 array operations.

1. Create an array `styles` with items “Jazz” and “Blues”.
2. Append “Rock-n-Roll” to the end.
3. Replace the value in the middle by “Classics”. Your code for finding the middle value should work for any arrays with odd length.

4. Strip off the first value of the array and show it.

5. 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
```

[To solution](#)

Calling in an array context

importance: 5

What is the result? Why?

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
})

arr[2](); // ?
```

[To solution](#)

Sum input numbers

importance: 4

Write 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 demo](#)

[To solution](#)

A maximal subarray

importance: 2

The 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.](#) ↗

[To solution](#)

Array methods

Arrays provide a lot of methods. To make things easier, in this chapter they are split into groups.

Add/remove items

We 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 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(str)` method is a swiss army knife for arrays. It can do everything: add, remove and insert elements.

The syntax is:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

It starts from the position `index`: 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
```

slice

The method `arr.slice ↗` is much simpler than similar-looking `arr.splice`.

The syntax is:

```
arr.slice(start, end)
```

It returns a new array where it copies all items start index `"start"` to `"end"` (not including `"end"`). Both `start` and `end` can be negative, in that case position from array end is assumed.

It works like `str.slice`, but makes subarrays instead of substrings.

For instance:

```
let str = "test";
let arr = ["t", "e", "s", "t"];

alert( str.slice(1, 3) ); // es
alert( arr.slice(1, 3) ); // e,s
```

```
alert( str.slice(-2) ); // st
alert( arr.slice(-2) ); // s,t
```

concat

The method `arr.concat`  joins the array with other arrays and/or 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 is an array or has `Symbol.isConcatSpreadable` property, then all its elements are copied. Otherwise, the argument itself is copied.

For instance:

```
let arr = [1, 2];

// merge arr with [3,4]
alert( arr.concat([3, 4])); // 1,2,3,4

// merge arr with [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

// merge arr with [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 (“spreads” them). Other objects, even if they look like arrays, added as a whole:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
//[1, 2, arrayLike]
```

...But if an array-like object has `Symbol.isConcatSpreadable` property, then 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

```

Searching in array

These are methods to search for something in an array.

`indexOf/lastIndexOf` and `includes`

The methods `arr.indexOf` ↗, `arr.lastIndexOf` ↗ and `arr.includes` ↗ have the same 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.lastIndexOf(item, from)` – same, but looks from right to left.
- `arr.includes(item, from)` – looks for `item` starting from index `from`, returns `true` if found.

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

```

Note that the methods use `==` comparison. So, if we look for `false`, it finds exactly `false` and not the zero.

If we want to check for inclusion, and don't want to know the exact index, then `arr.includes` is preferred.

Also, a very minor difference of `includes` is that it correctly handles `Nan`, unlike `indexOf/lastIndexOf`:

```

const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (should be 0, but == equality doesn't work for NaN)
alert( arr.includes(NaN) );// true (correct)

```

`find` and `findIndex`

Imagine we have an array of objects. How do we find an object with the specific condition?

Here the `arr.find` ↗ method comes in handy.

The syntax is:

```
let result = arr.find(function(item, index, array) {
    // should return true if the item is what we are looking for
});
```

The function is called repetitively for each element of the array:

- `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` a single-argument function `item => item.id == 1`. Other parameters of `find` are rarely used.

The [arr.findIndex ↗](#) method is essentially the same, but it returns the index where the element was found instead of the element itself.

filter

The `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 roughly the same as `find`, but it returns an array of matching elements:

```
let results = arr.filter(function(item, index, array) {
    // should return true if the item passes the filter
});
```

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

This section is about the methods transforming or reordering the array.

map

The `arr.map ↗` method is one of the most useful and often used.

The syntax is:

```
let result = arr.map(function(item, index, array) {  
  // returns the new value instead of item  
})
```

It calls the function for each element of the array and returns the array of results.

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 method `arr.sort ↗` sorts the array *in place*.

For instance:

```
let arr = [ 1, 2, 15 ];  
  
// the method reorders the content of arr (and returns it)  
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 and then compared. So, the lexicographic ordering is applied and indeed `"2" > "15"`.

To use our own sorting order, we need to supply a function of two arguments as the argument of `arr.sort()`.

The function should work like this:

```
function compare(a, b) {  
    if (a > b) return 1;  
    if (a == b) return 0;  
    if (a < b) return -1;  
}
```

For instance:

```
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 html elements or whatever. We have a set of *something*. 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 has a built-in implementation of sorting algorithm. We don't need to care how it exactly works (an optimized [quicksort ↗](#) 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 );  
});
```

The algorithm may compare an element multiple times in the process, but it tries to make as few comparisons as possible.

i 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
```

i 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 other, longer, version above.

reverse

The 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 join

Here's the situation from the 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(str)` does the reverse to `split`. It creates a string of `arr` items glued by `str` between them.

For instance:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';');
alert(str); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

When we need to iterate over an array – we can use `forEach`.

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(previousValue, item, index, arr) {
  // ...
}, initial);
```

The function is applied to the elements. You may notice the familiar arguments, starting from the 2nd:

- `item` – is the current array item.

- `index` – is its position.
- `arr` – is the array.

So far, like `forEach/map`. But there's one more argument:

- `previousValue` – is the result of the previous function call, `initial` for the first call.

The easiest way to grasp that is by example.

Here we get a sum of array in one line:

```
let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

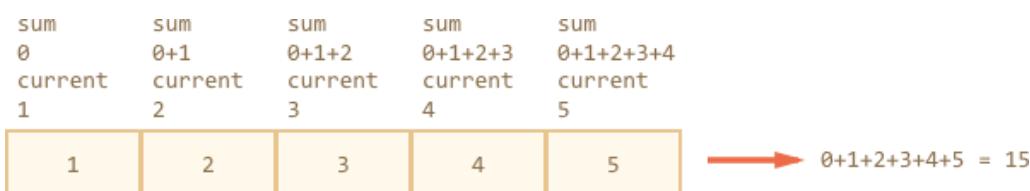
alert(result); // 15
```

Here we used the most common variant of `reduce` which uses only 2 arguments.

Let's see the details of what's going on.

1. 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 result is `1`.
2. On the second run, `sum = 1`, we add the second array element (`2`) to it and return.
3. 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	3
the third call	3	3	6
the fourth call	6	4	10
the fifth call	10	5	15

As we can see, 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.

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.

Array.isArray

Arrays do 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 []); // 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 instance, here we use an object method as a filter and `thisArg` comes in handy:

```
let user = {  
    age: 18,  
    younger(otherUser) {  
        return otherUser.age < this.age;  
    }  
};  
  
let users = [  
    {age: 12},  
    {age: 16},  
    {age: 32}  
];
```

```
// find all users younger than user
let youngerUsers = users.filter(user.younger, user);

alert(youngerUsers.length); // 2
```

In the call above, we use `user.younger` as a filter and also provide `user` as the context for it. If we didn't provide the context, `users.filter(user.younger)` would call `user.younger` as a standalone function, with `this=undefined`. That would mean an instant error.

Summary

A cheatsheet 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` delete `deleteCount` elements and insert `items`.
 - `slice(start, end)` – creates a new array, copies elements from position `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 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(func, initial)` – calculate a single value over the array by calling `func` for each element and passing an intermediate result between the calls.
- To iterate over elements:
 - `forEach(func)` – calls `func` for every element, does not return anything.
- Additionally:

- `Array.isArray(arr)` checks `arr` for being an array.

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)` ↗ checks 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`.

- `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).

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 than it seems.

Look through the cheatsheet 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 cheatsheet 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 borderLeftWidth

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.](#) ↗

[To solution](#)

Filter range

importance: 4

Write a function `filterRange(arr, a, b)` that gets an array `arr`, looks for elements between `a` and `b` in it and returns an array of them.

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.](#) ↗

[To solution](#)

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 ≤ arr[i] ≤ 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.](#) ↗

[To solution](#)

Sort in the reverse order

importance: 4

```
let arr = [5, 2, 1, -10, 8];

// ... your code to sort it in the reverse order

alert( arr ); // 8, 5, 2, 1, -10
```

[To solution](#)

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)
```

[To solution](#)

Create an extendable calculator

importance: 5

Create a constructor function `Calculator` that creates “extendable” calculator objects.

The task consists of two parts.

1. 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
```

2. 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 brackets 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. ↗

To solution

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
```

To solution

Map to objects

importance: 5

You 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 },
]
```

```
    { 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.

[To solution](#)

Sort objects

importance: 5

Write the function `sortByName(users)` that gets an array of objects with property `name` and sorts it.

For instance:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ john, pete, mary ];

sortByName(arr);

// now: [john, mary, pete]
alert(arr[1].name); // Mary
```

[To solution](#)

Shuffle an array

importance: 3

Write 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.

[To solution](#)

Get average age

importance: 4

Write the function `getAverageAge(users)` that gets an array of objects with property `age` and gets the average.

The formula for the average is `(age1 + age2 + ... + 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
```

[To solution](#)

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", ":-0"
];

alert( unique(strings) ); // Hare, Krishna, :-0
```

[Open a sandbox with tests.](#) ↗

[To solution](#)

Iterables

Iterable objects is a generalization of arrays. That's a concept that allows 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 iterable also. As we'll see, many built-in operators and methods rely on them.

If an object 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.iterator

We 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` 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).

1. 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`.
2. Onward, `for .. of` works *only with that returned object*.
3. When `for .. of` wants the next value, it calls `next()` on that object.
4. The result of `next()` must have the form `{done: Boolean, value: any}`, where `done=true` means that the iteration is finished, otherwise `value` must be the new value.

Here's the full implementation for `range`:

```
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 this iterator, 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: an important 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 it handles the whole 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, doable with some 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 = '𠮷';
for (let char of str) {
  alert( char ); // 𠮷, and then ⓘ
}
```

Calling an iterator explicitly

Normally, internals of iterables are hidden from the external code. There's a `for .. of` loop, that works, that's all it needs to know.

But to understand things a little bit deeper let's see how to create an iterator explicitly.

We'll iterate over a string the same way as `for .. of`, but with direct calls. This code gets a string iterator and calls 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

There are two official terms that 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.

Naturally, these properties can combine. 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) {}
```

What do they have in common? 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.

Array.from

There's a universal method `Array.from` ↫ that brings them together. It 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 there all items.

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` allows to provide an optional “mapping” function:

```
Array.from(obj[, mapFn, thisArg])
```

The second argument `mapFn` should be the function to apply to each element before adding to the array, and `thisArg` allows 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 = '𠮷';
// splits str into array of characters
let chars = Array.from(str);

alert(chars[0]); // 𠮷
alert(chars[1]); // 𠮷
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 = '𠮷𠮷';

let chars = []; // Array.from internally does the same loop
for (let char of str) {
  chars.push(char);
}

alert(chars);

```

...But 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 = '𠮷𠮷𩿵';

alert( slice(str, 1, 3) ); // 𩿵

// 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 the further iteration process.
 - An iterator must have the method named `next()` that returns an object `{done: Boolean, value: any}`, here `done:true` denotes the iteration end, 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` of 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.

Map, Set, WeakMap and WeakSet

Now we've learned about the following complex data structures:

- Objects for storing keyed collections.
- Arrays for storing ordered collections.

But that's not enough for real life. That's why `Map` and `Set` also exist.

Map

`Map` ↗ is a collection of keyed data items, just like an `Object`. But the main difference is that `Map` allows keys of any type.

The main methods 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 value by the key.
- `map.clear()` – clears 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 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 most notable and important `Map` features. For string keys, `Object` can be fine, but it would be difficult to replace the `Map` with a regular `Object` in the example above.

In the old times, before `Map` existed, people added unique identifiers to objects for that:

```
// we add the id field
let john = { name: "John", id: 1 };

let visitsCounts = {};

// now store the value by id
visitsCounts[john.id] = 123;

alert( visitsCounts[john.id] ); // 123
```

...But `Map` is much more elegant.

How `Map` compares keys

To test values 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');
```

Map from Object

When a `Map` is created, we can pass an array (or another iterable) with key-value pairs, like this:

```
// array of [key, value] pairs
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);
```

There is a built-in method `Object.entries(obj)` ↗ that returns an array of key/value pairs for an object exactly in that format.

So we can initialize a map from an object like this:

```
let map = new Map(Object.entries({  
    name: "John",  
    age: 30  
}));
```

Here, `Object.entries` returns the array of key/value pairs: `[["name", "John"], ["age", 30]]`. That's what `Map` needs.

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`:

```
recipeMap.forEach( (value, key, map) => {
  alert(`#${key}: ${value}`); // cucumber: 500 etc
});
```

Set

A `Set` is a collection of values, where each value may occur only once.

Its main methods are:

- `new Set(iterable)` – creates the set, optionally from an array of values (any iterable will do).
- `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.

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 `forEach` function in the `Set` has 3 arguments: a value, then *again a value*, and then the target object. Indeed, the same value appears in the arguments twice.

That's for compatibility with `Map` where `forEach` has three arguments.

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`.

WeakMap and WeakSet

`WeakSet` is a special kind of `Set` that does not prevent JavaScript from removing its items from memory. `WeakMap` is the same thing for `Map`.

As we know from the chapter [Garbage collection](#), JavaScript engine stores 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.

In a regular `Map`, it does not matter if we store an object as a key or as a value. It's kept in memory even if there are no more references to it.

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()
```

With the exception of `WeakMap`/`WeakSet`.

WeakMap/**WeakSet** does not prevent the object removal from the memory.

Let's start with `WeakMap`.

The first difference from `Map` is that its keys must be objects, not primitive values:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // works fine (object key)

weakMap.set("test", "Whoops"); // Error, because "test" is a primitive
```

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 is to be automatically deleted.

...And `WeakMap` does not support methods `keys()`, `values()`, `entries()`, we can not iterate over it. So there's really no way to receive all keys or values from it.

`WeakMap` has only the following methods:

- `weakMap.get(key)`
- `weakMap.set(key, value)`
- `weakMap.delete(key, value)`
- `weakMap.has(key)`

Why such a limitation? That's for technical reasons. If the object has lost all other references (like `john` in the code above), then it is to be deleted 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 the `WeakMap` is not known. The engine may have cleaned it up or not, or did it partially. For that reason, methods that access `WeakMap` as a whole are not supported.

Now where do we need such thing?

The idea of `WeakMap` is that we can store something for an object that exists only while the object exists. But we do not force the object to live by the mere fact that we store something for it.

```
weakMap.set(john, "secret documents");
// if john dies, secret documents will be destroyed
```

That's useful for situations when we have a main storage for the objects somewhere and need to keep additional information that is only relevant while the object lives.

Let's look at an example.

For instance, we have code that keeps a visit count for each user. The information is stored in a map: a user is the key and the visit count is the value. When a user leaves, we don't want to store their visit count anymore.

One way would be to keep track of leaving users and clean up the storage manually:

```
let john = { name: "John" };

// map: user => visits count
let visitsCountMap = new Map();

// john is the key for the map
visitsCountMap.set(john, 123);

// now john leaves us, we don't need him anymore
john = null;

// but it's still in the map, we need to clean it!
alert( visitsCountMap.size ); // 1
// it's also in the memory, because Map uses it as the key
```

Another way would be to use `WeakMap`:

```
let john = { name: "John" };

let visitsCountMap = new WeakMap();

visitsCountMap.set(john, 123);

// now john leaves us, we don't need him anymore
```

```

john = null;

// there are no references except WeakMap,
// so the object is removed both from the memory and from visitsCountMap automatically

```

With a regular `Map`, cleaning up after a user has left becomes a tedious task: we not only need to remove the user from its main storage (be it a variable or an array), but also need to clean up the additional stores like `visitsCountMap`. And it can become cumbersome in more complex cases when users are managed in one place of the code and the additional structure is at another place and is getting no information about removals.

`WeakMap` can make things simpler, because it is cleaned up automatically. The information in it like visits count in the example above lives only while the key object exists.

`WeakSet` 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.

For instance, we can use it to keep track of whether an item is checked:

```

let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

// fill it with array elements (3 items)
let unreadSet = new WeakSet(messages);

// we can use unreadSet to see whether a message is unread
alert(unreadSet.has(messages[1])); // true
// remove it from the set after reading
unreadSet.delete(messages[1]); // true

// and when we shift our messages history, the set is cleaned up automatically
messages.shift();
// no need to clean unreadSet, it now has 2 items
// unfortunately, there's no method to get the exact count of items, so can't show it

```

The most notable limitation of `WeakMap` and `WeakSet` is the absence of iterations, and inability to get all current content. That may appear inconvenient, but actually 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.

Summary

- `Map` – is a collection of keyed values.

The differences from a regular `Object`:

- Any keys, objects can be keys.
- Iterates in the insertion order.
- Additional convenient methods, the `size` property.
- `Set` – is a collection of unique values.
 - Unlike an array, does not allow to reorder elements.
 - Keeps the insertion order.
- `WeakMap` – a variant of `Map` that allows only objects as keys and removes them once they become inaccessible by other means.
 - It does not support operations on the structure as a whole: no `size`, no `clear()`, no iterations.
- `WeakSet` – is a variant of `Set` that only stores objects and removes them once they become inaccessible by other means.
 - Also does not support `size/clear()` and iterations.

`WeakMap` and `WeakSet` are used as “secondary” data structures in addition to the “main” object storage. Once the object is removed from the main storage, if it is only found in the `WeakMap/WeakSet`, it will be cleaned up automatically.

✓ Tasks

Filter unique array members

importance: 5

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 values = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-0"
];
alert( unique(values) ); // Hare, Krishna, :-0
```

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. ↗](#)

[To solution](#)

Filter anagrams

importance: 4

Anagrams [↗](#) 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 solution](#)

Iterable keys

importance: 5

We want to get an array of `map.keys()` and go on working with it (apart from the map itself).

But there's a problem:

```
let map = new Map();  
  
map.set("name", "John");  
  
let keys = map.keys();  
  
// Error: numbers.push is not a function  
keys.push("more");
```

Why? How can we fix the code to make `keys.push` work?

[To solution](#)

Store "unread" flags

importance: 5

There'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 you could use to store information whether the message "have 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 directly. If they are managed by someone else's code, then adding extra properties to them may have bad consequences.

[To solution](#)

Store read dates

importance: 5

There'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, once again, should disappear if the message is gone.

[To solution](#)

Object.keys, values, entries

Let'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 (except `arr.values()`)

Plain objects also support similar methods, but the syntax is a bit different.

Object.keys, values, entries

For 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.

...But please note the distinctions (compared to map for example):

	Map	Object
Call syntax	<code>map.keys()</code>	<code>Object.keys(obj)</code> , but not <code>obj.keys()</code>
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 `order` that implements its own `order.values()` method. And we still can call `Object.values(order)` 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.getOwnPropertySymbols()` that returns an array of only symbolic keys. Also, the method `Reflect.ownKeys(obj)` returns *all* keys.

Tasks

Sum the properties

importance: 5

There 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.](#)

[To solution](#)

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.](#) ↗

[To solution](#)

Destructuring assignment

The two most used data structures in JavaScript are `Object` and `Array`.

Objects allow us to pack many pieces of information into a single entity and arrays allow us to store ordered collections. So we can make an object or an array and handle it as a single entity, or maybe pass it to a function call.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes they are more convenient. Destructuring also works great with complex functions that have a lot of parameters, default values, and soon we’ll see how these are handled too.

Array destructuring

An example of how the array is destructured into variables:

```
// we have an array with the name and surname
let arr = ["Ilya", "Kantor"]

// destructuring assignment
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname); // Kantor
```

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] = "Ilya Kantor".split(' ');
```

i “Destructuring” does not mean “destructive”.

It's called “destructuring assignment,” because it “destructurizes” 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];
```

i Ignore first elements

Unwanted elements of the array can also be thrown away via an extra comma:

```
// first and second elements are not needed  
let [, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
  
alert( title ); // Consul
```

In the code above, although the first and second elements of the array are skipped, the third one is assigned to `title`, and the rest are also skipped.

i 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]);
```

i Assign to anything at the left-side

We can use any “assignables” at the left side.

For instance, an object property:

```
let user = {};  
[user.name, user.surname] = "Ilya Kantor".split(' ');\n\nalert(user.name); // Ilya
```

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  
}
```

...And the same for a map:

```
let user = new Map();  
user.set("name", "John");  
user.set("age", "30");  
  
for (let [key, value] of user.entries()) {  
    alert(` ${key}: ${value}`); // name: John, then age: 30  
}
```

The rest ‘...’

If we want not just to get first values, but 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"];  
  
alert(name1); // Julius  
alert(name2); // Caesar  
  
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.

Default values

If there are fewer values in the array than variables in the assignment, there will be no error. Absent values are considered undefined:

```
let [firstName, surname] = [];
```

```
alert(firstName); // 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. But it will run only for the missing one:

```
// runs only prompt for surname
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name);    // Julius (from array)
alert(surname); // whatever prompt gets
```

Object destructuring

The destructuring assignment also works with objects.

The basic syntax is:

```
let {var1, var2} = {var1:..., var2...}
```

We have an existing object at the right side, that we want to split into variables. The left side contains a “pattern” for corresponding properties. In the simple 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 of properties 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, `options.width` to go into the variable named `w`, then we can set it 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.

The code below asks for `width`, but not the `title`.

```
let options = {
  title: "Menu"
```

```
};

let {width = prompt("width?"), title = prompt("title?")} = options;

alert(title); // Menu
alert(width); // (whatever you 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
```

The rest operator

What if the object has more properties than we have variables? Can we take some and then assign the “rest” somewhere?

The specification for using the rest operator (three dots) here is almost in the standard, but most browsers do not support it yet.

It looks like this:

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};

let {title, ...rest} = options;

// now title="Menu", rest={height: 200, width: 100}
alert(rest.height); // 200
alert(rest.width); // 100
```

Gotcha without `let`

In the examples above variables were declared right before the assignment: `let { ... } = { ... }`. Of course, we could use existing variables too. 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 );
}
```

To show JavaScript that it's not a code block, we can wrap the whole assignment in brackets `(...)`:

```
let title, width, height;

// okay now
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

Nested destructuring

If an object or an array contain other 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 at the left side of the assignment has the same structure:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true    // something extra that we will not destruct
};
```

```
// destructuring assignment on 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
```

The whole `options` object except `extra` that was not mentioned, is assigned to corresponding variables.

```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
}

let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}
```

Finally, we have `width`, `height`, `item1`, `item2` and `title` from the default value.

That often happens with destructuring assignments. We have a complex object with many properties and want to extract only what we need.

Even here it happens:

```
// take size as a whole into a variable, ignore the rest
let { size } = options;
```

Smart function parameters

There are times when a function may have 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?

```
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 destructure 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 syntax is the same as for a destructuring assignment:

```
function({
  incomingProperty: parameterName = defaultValue
```

```
...  
})
```

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({});  
  
showMenu(); // this would give an error
```

We can fix this by making `{}` the default value for the whole destructuring thing:

```
// simplified parameters a bit for clarity  
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 object syntax:

```
let {prop : varName = default, ...} = 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.

- The 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`.

- For more complex cases, the left side must have the same structure as the right one.

Tasks

Destructuring assignment

importance: 5

We 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 absent)

The values after the assignment should be:

```
let user = { name: "John", years: 30 };  
  
// your code to the left side:  
// ... = user  
  
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

[To solution](#)

The maximal salary

importance: 5

There 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.](#) ↗

[To solution](#)

Date and time

Let'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, or to measure time, or just to print out the current date.

Creation

To 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 );
```

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).

`new Date(datestring)`

If there is a single argument, and it's a string, then it is parsed with the `Date.parse` algorithm (see below).

```
let date = new Date("2017-01-26");
alert(date); // Thu Jan 26 2017 ...
```

```
new Date(year, month, date, hours, minutes, seconds, ms)
```

Create the date with the given components in the local time zone. Only two first arguments are obligatory.

Note:

- The `year` must have 4 digits: `2013` is okay, `98` is not.
- 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); // // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // the same, hours etc are 0 by default
```

The minimal 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 many methods to access the year, month and so on from the `Date` object. But they can be easily remembered when categorized.

[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 `getFullYear()`, but `getYear()`

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 the local time zone and UTC, 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 components

The 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.
```

Autocorrection

The *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 diff

When 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 counting

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // done

alert(`The loop took ${end - start} ms`);
```

Date.now()

If we only want to measure the difference, 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
```

Benchmarking

If 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?

```
// 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 around 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)` the 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.

Here's the code example:

```
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(upperSlice) and bench(upperLoop) 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. 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 <http://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` that 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.

Also, 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 34731.2600000001ms ago"  
// .26 is microseconds (260 microseconds)  
// more than 3 digits after the decimal point are precision errors, but only the first 3 are c
```

Node.JS has `microtime` module and other ways. Technically, any device and environment allows to get more precision, it's just not in `Date`.

Tasks

Create a date

importance: 5

Create a `Date` object for the date: Feb 20, 2012, 3:12am. The time zone is local.

Show it using `alert`.

[To solution](#)

Show a weekday

importance: 5

Write 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.](#)

[To solution](#)

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.](#)

[To solution](#)

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 also work over months/years reliably:

```
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.](#)

[To solution](#)

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.](#) ↗

[To solution](#)

How many seconds has passed today?

importance: 5

Write a function `getSecondsToday()` that returns the number of seconds from the beginning of today.

For instance, if now `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”.

[To solution](#)

How many seconds till tomorrow?

importance: 5

Create 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.

[To solution](#)

Format the relative date

importance: 4

Write 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 31.12.2016, 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

[Open a sandbox with tests.](#) ↗

[To solution](#)

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. And, if we're sending the object over a network, then we also need to supply the code to "read" our object on the receiving side.

Luckily, there's no need to write the code to handle all this. The task has been solved already.

JSON.stringify

The [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'],
  wife: 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"],
  "wife": null
}
*/
```

The method `JSON.stringify(student)` takes the object and converts it into a string.

The resulting `json` string is a called *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.

Natively supported JSON types are:

- 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 cross-language specification, so some JavaScript-specific object properties are skipped by `JSON.stringify`.

Namely:

- Function properties (methods).
- Symbolic properties.
- 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:

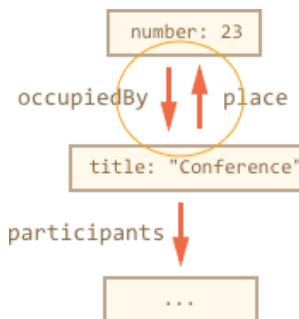
```
let room = {
  number: 23
};

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: replacer

The 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 participants are empty, because `name` is not in the list.

Let's include 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.

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}`); // to see what replacer gets
  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
*/

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 the whole object if necessary.

Formatting: spacer

The third argument of `JSON.stringify(value, replacer, spaces)` 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 `spacer` argument is used exclusively for a nice output.

Here `spacer = 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 `spaces` 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`:

```

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 for the nested 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 user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

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 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 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  
};
```

[To solution](#)

Exclude backreferences

importance: 5

In simple cases of circular references, we can exclude an offending property from serialization by its name.

But sometimes there are many backreferences. And names may be used both in circular references and normal properties.

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}
}
*/
```

[To solution](#)

Advanced working with functions

Recursion and stack

Let'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 thinking

For 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.

1. 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
```

2. 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
```

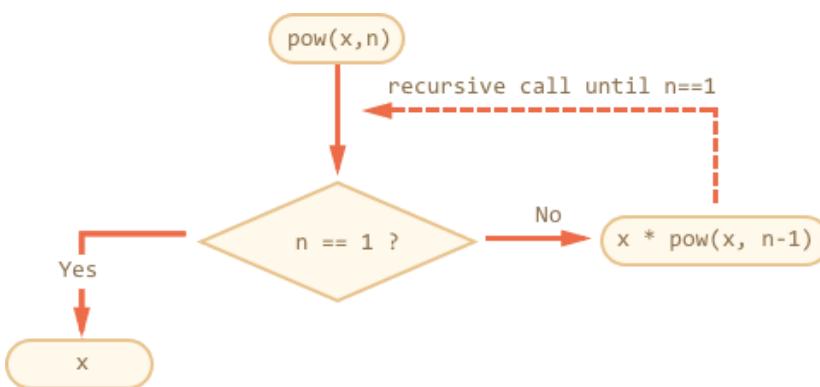
Please note how the recursive variant is fundamentally different.

When `pow(x, n)` is called, the execution splits into two branches:

```
if n==1 = x  
/  
pow(x, n) =  
 \  
else     = x * pow(x, n - 1)
```

1. If `n == 1`, then everything is trivial. It is called *the base of recursion*, because it immediately produces the obvious result: `pow(x, 1)` equals `x`.
2. Otherwise, we can represent `pow(x, n)` as `x * pow(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 `pow(2, 4)` the recursive variant does these steps:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `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.

i Recursion is usually shorter

A recursive solution is usually shorter than an iterative one.

Here we can rewrite the same using the ternary `?` operator instead of `if` to make `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 make sure about 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 stack

Now let's examine how recursive calls work. For that we'll look under the hood of functions.

The information about a function run 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 }** call: pow(2, 3)

That's when the function starts to execute. The condition `n == 1` is false, 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 }** call: pow(2, 3)

To calculate `x * 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:

1. The current context is “remembered” on top of the stack.
2. The new context is created for the subcall.
3. 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 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: 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. Here in the picture we use the word “line”, but of course it's more precise.

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 }** call: `pow(2, 1)`
- **Context: { x: 2, n: 2, at line 5 }** call: `pow(2, 2)`
- **Context: { x: 2, n: 3, at line 5 }** call: `pow(2, 3)`

There are 2 old contexts now and 1 currently running for `pow(2, 1)`.

The exit

During 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 }** call: `pow(2, 2)`
- **Context: { x: 2, n: 3, at line 5 }** call: `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 }** call: `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 contexts 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 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: 600
        }
    ],
    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 the 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? Should we stop on level 3 or make 4 levels of loops? 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:

1. Either it's a “simple” department with an *array of people* – then we can sum the salaries in a simple loop.
2. Or it's an *object with N subdepartments* – then we can make `N` recursive calls to get the sum for each of the subdeps and combine the results.

The (1) is the base of recursion, the trivial case.

The (2) 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: 600}],
  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)); // 6700

```

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.

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:

```

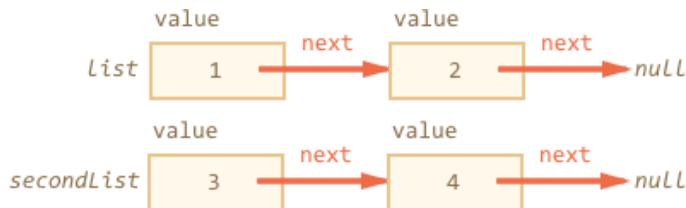
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
  
```

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:

```

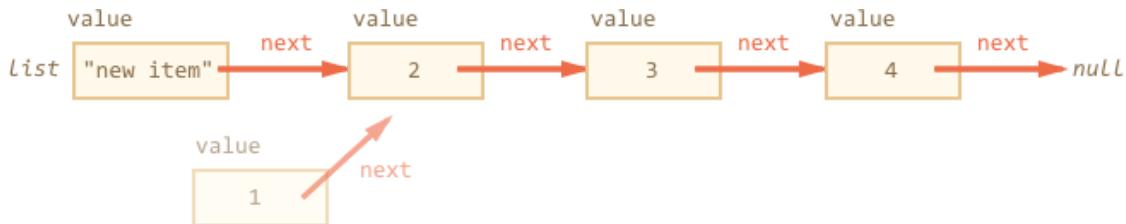
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// 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 `N`th 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.

Sometimes it's worth to add another variable named `tail` to track the last element of the list (and update it when adding/removing elements from the end). For large sets of elements the speed difference versus arrays is huge.

Summary

Terms:

- *Recursion* is a programming term that means a “self-calling” function. Such functions can be used to solve certain 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 branch 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.

✓ Tasks

Sum all numbers till the given one

importance: 5

Write a function `sumTo(n)` that calculates the sum of numbers `1 + 2 + ... + 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:

1. Using a for loop.
2. Using a recursion, cause `sumTo(n) = n + sumTo(n-1)` for `n > 1`.
3. 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(1000000)`?

[To solution](#)

Calculate factorial

importance: 4

The [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) * ... * 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: $3! = 3 * 2! = 3 * 2 * 1! = 6$

[To solution](#)

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.

[To solution](#)

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?

[To solution](#)

Output a single-linked list in the reverse order

importance: 5

Output 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.

[To solution](#)

Rest parameters and spread operator

Many 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, more importantly, how to feel comfortable working with such functions and arrays.

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.

The rest parameters can be mentioned in a function definition with three dots `...`. They 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 has no sense:

```
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, no matter their total number.

And it still works, we can use it today.

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.

i 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 operator

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 operator 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 operator 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 operator 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 operator, but any iterable will do.

For instance, here we use the spread operator to turn the string into array of characters:

```
let str = "Hello";

alert(...str); // H,e,l,l,o
```

The spread operator 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 operator operates only on iterables.

So, for the task of turning something into an array, `Array.from` tends to be more universal.

Summary

When we see `"..."` in the code, it is either rest parameters or the spread operator.

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 operator” and expands an array into a list.

Use patterns:

- Rest parameters are used to create functions that accept any number of arguments.
- The spread operator 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.

Closure

JavaScript is a very function-oriented language. It gives us a lot of freedom. A function can be created at one moment, then copied to another variable or passed as an argument to another function and called from a totally different place later.

We know that a function can access variables outside of it; this feature is used quite often.

But what happens when an outer variable changes? Does a function get the most recent value or the one that existed when the function was created?

Also, what happens when a function travels to another place in the code and is called from there – does it get access to the outer variables of the new place?

Different languages behave differently here, and in this chapter we cover the behaviour of JavaScript.

A couple of questions

Let's consider two situations to begin with, and then study the internal mechanics piece-by-piece, so that you'll be able to answer the following questions and more complex ones in the future.

- The 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?

2. The function `makeworker` 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? "Pete" (name where created) or "John" (name where called)?
```

Lexical Environment

To understand what's going on, let's first discuss what a “variable” actually is.

In JavaScript, every running function, code block, and the script as a whole have an associated object known as the *Lexical Environment*.

The Lexical Environment object consists of two parts:

1. *Environment Record* – an object that has all local variables as its properties (and some other information like the value of `this`).
2. A reference to the *outer lexical environment*, usually the one associated with the code lexically right outside of it (outside of the current curly brackets).

So, 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 the Lexical Environment”.

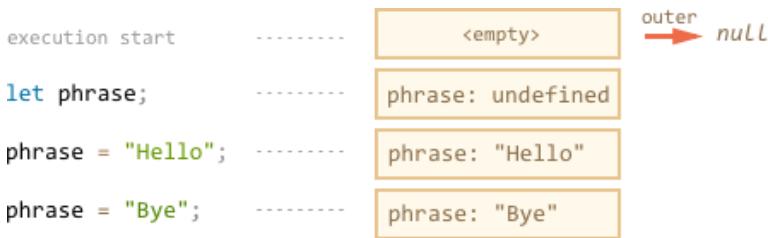
For instance, in this simple code, there is only one Lexical Environment:

```
let phrase = "Hello"; ----- phrase: "Hello" outer null  
alert(phrase);
```

This is a so-called global Lexical Environment, associated with the whole script. For browsers, all `<script>` tags share the same global environment.

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, so it points to `null`.

Here's the bigger picture of how `let` variables work:



Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

1. When the script starts, the Lexical Environment is empty.
2. The `let phrase` definition appears. It has been assigned no value, so `undefined` is stored.
3. `phrase` is assigned a value.
4. `phrase` refers to a new value.

Everything looks simple for now, right?

To summarize:

- 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.

Function Declaration

Function Declarations are special. Unlike `let` variables, they are processed not when the execution reaches them, but when a Lexical Environment is created. For the global Lexical Environment, it means the moment when the script is started.

That is why we can call a function declaration before it is defined.

The code below demonstrates that the Lexical Environment is non-empty from the beginning. It has `say`, because that's a Function Declaration. And later it gets `phrase`, declared with `let`:

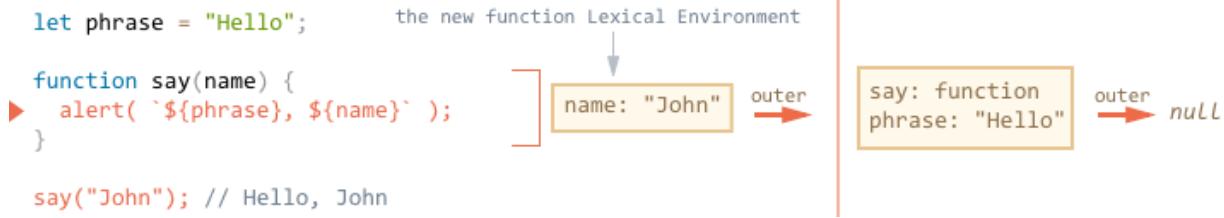


Inner and outer Lexical Environment

During the call, `say()` uses an outer variable, so let's look at the details of what's going on.

First, when a function runs, a new function Lexical Environment is created automatically. That's a general rule for all functions. That Lexical Environment is used to store local variables and parameters of the call.

Here's the picture of Lexical Environments when the execution is inside `say("John")`, 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 variable: `name`, the function argument. We called `say("John")`, so the value of `name` is `"John"`.
- The outer Lexical Environment is the global Lexical Environment.

The inner Lexical Environment has the `outer` reference to the outer one.

When code wants to access a variable – it is first searched for in the inner Lexical Environment, then in the outer one, then the more outer one and so on until the end of the chain.

If a variable is not found anywhere, that's an error in strict mode. Without use `strict`, an assignment to an undefined variable creates a new global variable, for backwards compatibility.

Let's see how the search proceeds in our example:

- When the `alert` inside `say` wants to access `name`, it finds it immediately in the function Lexical Environment.
- When it wants to access `phrase`, then there is no `phrase` locally, so it follows the `outer` reference and finds it globally.



Now we can give the answer to the first question from the beginning of the chapter.

A function gets outer variables as they are now; it uses the most recent values.

That's because of the described mechanism. Old variable values are not saved anywhere. When a function wants them, it takes the current values from its own or an outer Lexical Environment.

So the answer to the first question is `Pete`:

```

let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

```

```
name = "Pete"; // (*)
```

```
sayHi(); // Pete
```

The execution flow of the code above:

1. The global Lexical Environment has `name: "John"`.
2. At the line `(*)` the global variable is changed, now it has `name: "Pete"`.
3. When the function `sayHi()`, is executed and takes `name` from outside. Here that's from the global Lexical Environment where it's already `"Pete"`.

i One call – one Lexical Environment

Please note that a new function Lexical Environment is created each time a function runs.

And if a function is called multiple times, then each invocation will have its own Lexical Environment, with local variables and parameters specific for that very run.

i Lexical Environment is a specification object

“Lexical Environment” is a specification object. 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, but the visible behavior should be as described.

Nested functions

A 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.

What’s more interesting, a nested function can be returned: either as a property of a new object (if the outer function creates an object with methods) 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.

An example with the constructor function (see the chapter [Constructor, operator "new"](#)):

```
// constructor function returns a new object
function User(name) {

    // the object method is created as a nested function
    this.sayHi = function() {
        alert(name);
    };
}

let user = new User("John");
user.sayHi(); // the method code has access to the outer "name"
```

An example with returning a function:

```
function makeCounter() {
    let count = 0;

    return function() {
        return count++; // has access to the outer counter
    };
}

let counter = makeCounter();

alert(counter()); // 0
alert(counter()); // 1
alert(counter()); // 2
```

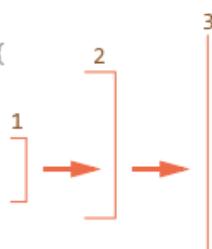
Let's go on with the `makeCounter` example. It creates the “counter” function that returns the next number on each invocation. Despite being simple, slightly modified variants of that code have practical uses, for instance, as a [pseudorandom number generator ↗](#), and more. So the example is not as artificial as it may appear.

How does the counter work internally?

When the inner function runs, the variable in `count++` is searched from inside out. For the example above, the order will be:

```
function makeCounter() {
    let count = 0;

    return function() {
        return count++;
    };
}
```



1. The locals of the nested function...
2. The variables of the outer function...
3. And so on until it reaches global variables.

In this example `count` is found on step 2. When an outer variable is modified, it's changed where it's found. So `count++` finds the outer variable and increases it in the Lexical Environment where it belongs. Like if we had `let count = 1`.

Here are two questions to consider:

1. Can we somehow reset the `counter` from the code that doesn't belong to `makeCounter`? E.g. after `alert` calls in the example above.
2. If we call `makeCounter()` multiple times – it returns many `counter` functions. Are they independent or do they share the same `count`?

Try to answer them before you continue reading.

...

All done?

Okay, let's go over the answers.

1. There is no way. The `counter` is a local function variable, we can't access it from the outside.
2. For every call to `makeCounter()` a new function Lexical Environment is created, with its own `counter`. So the resulting `counter` functions are independent.

Here's the demo:

```
function makeCounter() {  
  let count = 0;  
  return function() {  
    return count++;  
  };  
}  
  
let counter1 = makeCounter();  
let counter2 = makeCounter();  
  
alert(counter1()); // 0  
alert(counter1()); // 1  
  
alert(counter2()); // 0 (independent)
```

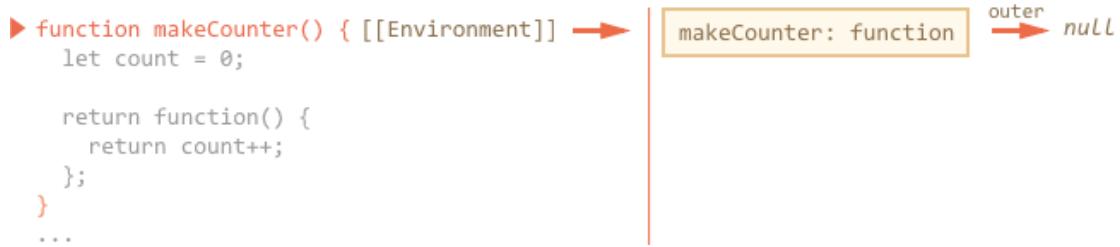
Hopefully, the situation with outer variables is quite clear for you now. But in more complex situations a deeper understanding of internals may be required. So let's dive deeper.

Environments in detail

Now that you understand how closures work generally, we can descend to the very nuts and bolts.

Here's what's going on in the `makeCounter` example step-by-step, follow it to make sure that you understand everything. Please note the additional `[[Environment]]` property that we didn't cover yet.

1. When the script has just started, there is only global Lexical Environment:



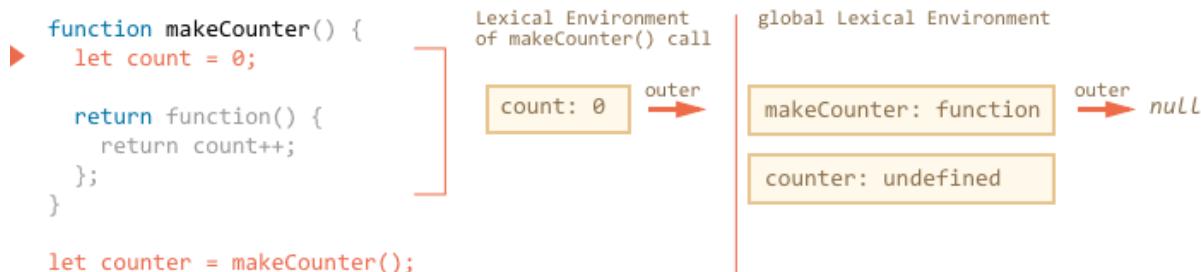
At that starting moment there is only `makeCounter` function, because it's a Function Declaration. It did not run yet.

All functions “on birth” receive a hidden property `[[Environment]]` with a reference to the Lexical Environment of their creation. We didn’t talk about it yet, but that’s how the function knows where it was made.

Here, `makeCounter` is created in the global Lexical Environment, so `[[Environment]]` keeps a reference to it.

In other words, a function is “imprinted” with a reference to the Lexical Environment where it was born. And `[[Environment]]` is the hidden function property that has that reference.

2. The code runs on, the new global variable `counter` is declared and for its value `makeCounter()` is called. Here’s a snapshot of the moment when the execution is on the first line inside `makeCounter()`:



At the moment of the call of `makeCounter()`, the Lexical Environment is created, to hold its variables and arguments.

As all Lexical Environments, it stores two things:

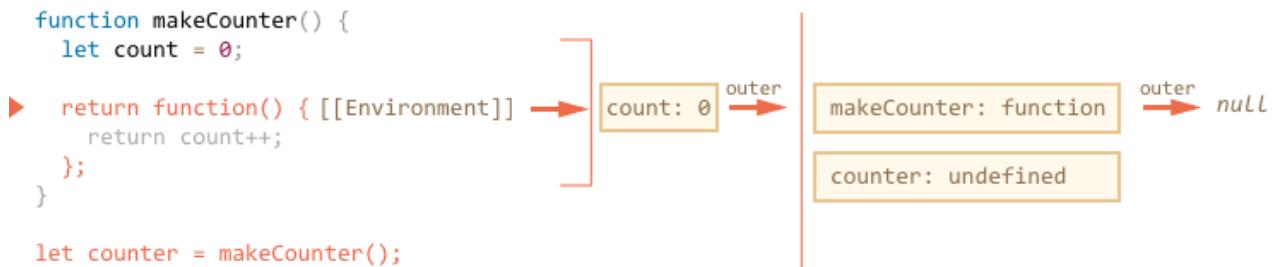
1. An Environment Record with local variables. In our case `count` is the only local variable (appearing when the line with `let count` is executed).
2. The outer lexical reference, which is set to `[[Environment]]` of the function. Here `[[Environment]]` of `makeCounter` references the global Lexical Environment.

So, now we have two Lexical Environments: the first one is global, the second one is for the current `makeCounter` call, with the outer reference to global.

3. During the execution of `makeCounter()`, a tiny nested function is created.

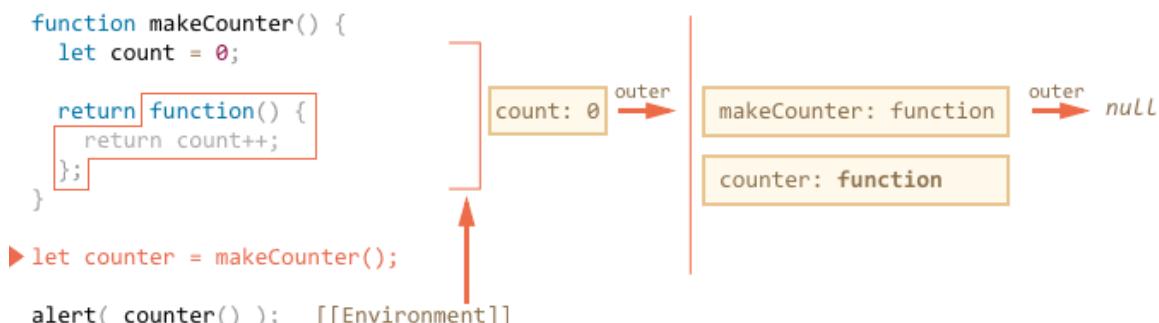
It doesn’t matter whether the function is created using Function Declaration or Function Expression. All functions get the `[[Environment]]` property that references the Lexical Environment in which they were made. So our new tiny nested function gets it as well.

For our new nested function the value of `[[Environment]]` is the current Lexical Environment of `makeCounter()` (where it was born):



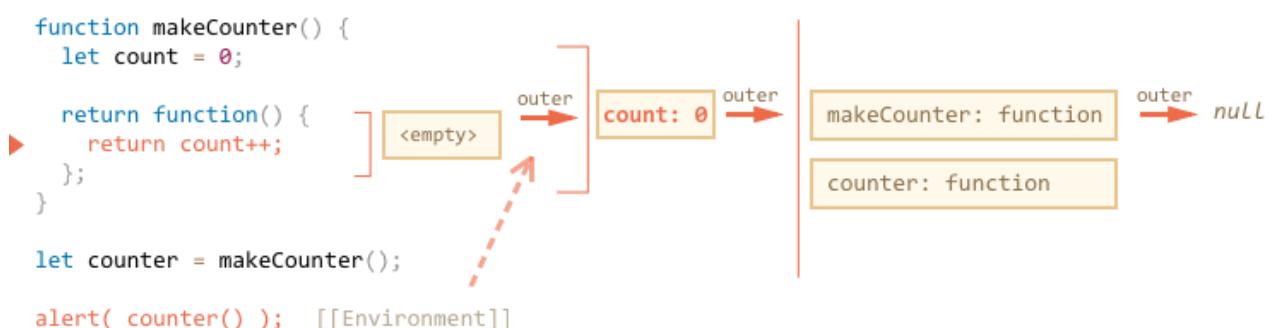
Please note that on this step the inner function was created, but not yet called. The code inside `function() { return count++; }` is not running; we're going to return it soon.

- As the execution goes on, the call to `makeCounter()` finishes, and the result (the tiny nested function) is assigned to the global variable `counter`:



That function has only one line: `return count++`, that will be executed when we run it.

- When the `counter()` is called, an “empty” Lexical Environment is created for it. It has no local variables by itself. But the `[[Environment]]` of `counter` is used as the outer reference for it, so it has access to the variables of the former `makeCounter()` call where it was created:



Now if it accesses a variable, it first searches its own Lexical Environment (empty), then the Lexical Environment of the former `makeCounter()` call, then the global one.

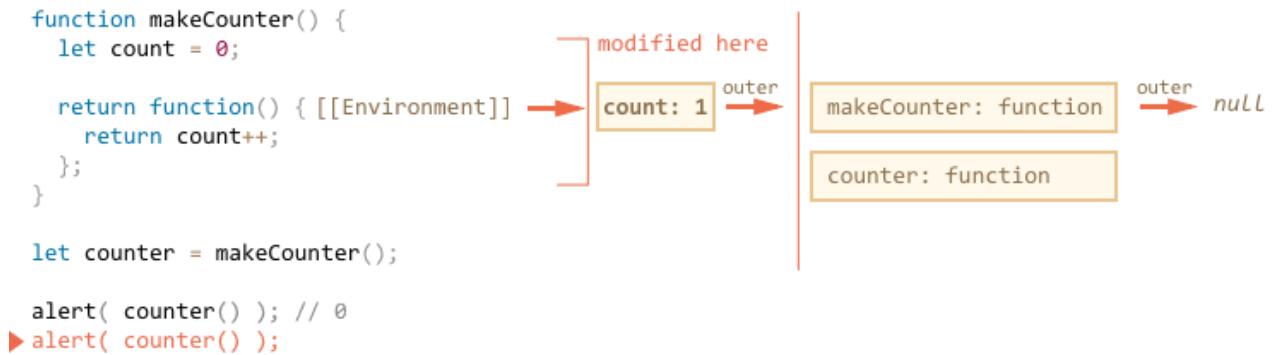
When it looks for `count`, it finds it among the variables `makeCounter`, in the nearest outer Lexical Environment.

Please note how memory management works here. Although `makeCounter()` call finished some time ago, its Lexical Environment was retained in memory, because there's a nested function with `[[Environment]]` referencing it.

Generally, a Lexical Environment object lives as long as there is a function which may use it. And only when there are none remaining, it is cleared.

- The call to `counter()` not only returns the value of `count`, but also increases it. Note that the modification is done “in place”. The value of `count` is modified exactly in the

environment where it was found.

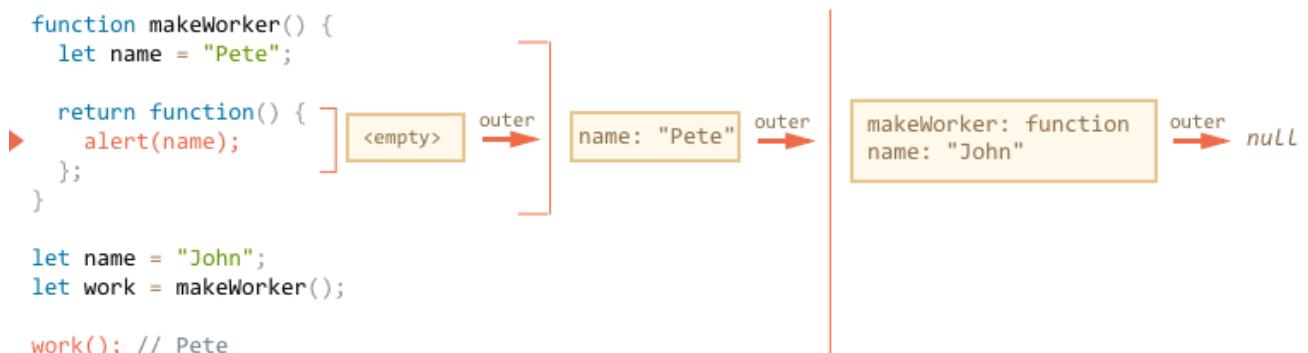


So we return to the previous step with the only change – the new value of `count`. The following calls all do the same.

7. Next `counter()` invocations do the same.

The answer to the second question from the beginning of the chapter should now be obvious.

The `work()` function in the code below uses the `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 it would be "John".

i Closures

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 exclusion, to be covered in [The "new Function" syntax](#)).

That is: they automatically remember where they were created using a hidden `[[Environment]]` property, and all of them 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 few more words about technical details: the `[[Environment]]` property and how Lexical Environments work.

Code blocks and loops, IIFE

The examples above concentrated on functions. But Lexical Environments also exist for code blocks `{ . . . }`.

They are created when a code block runs and contain block-local variables. Here are a couple of examples.

If

In the example below, when the execution goes into `if` block, the new “if-only” Lexical Environment is created for it:

```
let phrase = "Hello";  
  
if (true) {  
  let user = "John";  
  ▶ alert(`${phrase}, ${user}`);  
}  
  
alert(user); // Error, can't see such variable!
```

The diagram illustrates the creation of a new lexical environment within an `if` block. It shows two environments side-by-side. On the left, the global environment contains `phrase: "Hello"`. Inside the `if` block, a new lexical environment is created, containing `user: "John"`. An arrow labeled "outer" points from this inner environment to the `phrase` variable in the outer environment. The `user` variable in the inner environment is highlighted with a red box. On the right, the global environment contains `phrase: "Hello"`, which is also highlighted with a red box, and an arrow labeled "outer" points to `null`.

The new Lexical Environment gets the enclosing one as the outer reference, so `phrase` can be found. But all variables and Function Expressions declared inside `if` reside in that Lexical Environment and can't be seen from the outside.

For instance, after `if` finishes, the `alert` below won't see the `user`, hence the error.

For, while

For a loop, every iteration has a separate Lexical Environment. If a variable is declared in `for`, then it's also local to that Lexical Environment:

```
for (let i = 0; i < 10; i++) {  
  // Each loop has its own Lexical Environment  
  // {i: value}  
}  
  
alert(i); // Error, no such variable
```

That's actually an exception, because `let i` is visually outside of `{ . . . }`. But in fact each run of the loop has its own Lexical Environment with the current `i` in it.

After the loop, `i` is not visible.

Code blocks

We also can use a “bare” code block `{...}` to isolate variables into a “local scope”.

For instance, in a web browser all scripts share the same global area. So if we create a global variable in one script, it becomes available to others. But that becomes a source of conflicts if two scripts use the same variable name and overwrite each other.

That may happen if the variable name is a widespread word, and script authors are unaware of each other.

If we'd like to avoid that, we can use a code block to isolate the whole script or a part of it:

```
{  
  // do some job with local variables that should not be seen outside  
  
  let message = "Hello";  
  
  alert(message); // Hello  
}  
  
alert(message); // Error: message is not defined
```

The code outside of the block (or inside another script) doesn't see variables inside the block, because the block has its own Lexical Environment.

IIFE

In old scripts, one can find so-called “immediately-invoked function expressions” (abbreviated as IIFE) used for this purpose.

They look like this:

```
(function() {  
  
  let 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 brackets `(function ...)`, because when JavaScript meets `"function"` in the main code flow, it understands it as the start of a Function Declaration. But a Function Declaration must have a name, so there will be an error:

```
// Error: Unexpected token (
function() { // <-- JavaScript cannot find function name, meets ( and gives error  
  
  let message = "Hello";  
  
  alert(message); // Hello  
  
}();
```

We can say “okay, let it be so Function Declaration, let's add a name”, but it won't work. JavaScript does not allow Function Declarations to be called immediately:

```
// syntax error because of brackets below
function go() {

}(); // <-- can't call Function Declaration immediately
```

So, round brackets are needed 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 are other ways to tell JavaScript that we mean Function Expression:

```
// Ways to create IIFE

(function() {
  alert("Brackets around the function");
})();

(function() {
  alert("Brackets 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.

Garbage collection

Lexical Environment objects that we've been talking about are subject to the same memory management rules as regular values.

- Usually, Lexical Environment is cleaned up after the function run. For instance:

```
function f() {
  let value1 = 123;
  let value2 = 456;
}

f();
```

Here two values are technically the properties of the Lexical Environment. But after `f()` finishes that Lexical Environment becomes unreachable, so it's deleted from the memory.

- ...But if there's a nested function that is still reachable after the end of `f`, then its `[[Environment]]` reference keeps the outer lexical environment alive as well:

```

function f() {
  let value = 123;

  function g() { alert(value); }

  return g;
}

let g = f(); // g is reachable, and keeps the outer lexical environment in memory

```

- Please note that if `f()` is called many times, and resulting functions are saved, then the corresponding Lexical Environment objects will also be retained in memory. All 3 of them in the code below:

```

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
//           LE   LE   LE
let arr = [f(), f(), f()];

```

- A Lexical Environment object dies when it becomes unreachable: when no nested functions remain that reference it. In the code below, after `g` becomes unreachable, the `value` is also cleaned from memory:

```

function f() {
  let value = 123;

  function g() { alert(value); }

  return g;
}

let g = f(); // while g is alive
// there corresponding Lexical Environment lives

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 easy to see that an outer variable is not used – it is removed.

An important side effect in V8 (Chrome, 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();
```

⚠ See ya!

This feature of V8 is good to know. If you are debugging with Chrome/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 always can check for it by running the examples on this page.

✔ Tasks

Are counters independent?

importance: 5

Here 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()); // ?
```

[To solution](#)

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()); // ?
```

[To solution](#)

Function in if

Look at the code. What will be result of the call at the last line?

```
let phrase = "Hello";  
  
if (true) {  
  let user = "John";  
  
  function sayHi() {  
    alert(` ${phrase}, ${user}`);  
  }  
}  
  
sayHi();
```

[To solution](#)

Sum with closures

importance: 4

Write function `sum` that works like this: `sum(a)(b) = a+b`.

Yes, exactly this way, via double brackets (not a mistype).

For instance:

```
sum(1)(2) = 3  
sum(5)(-1) = 4
```

[To solution](#)

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. ↗

[To solution](#)

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.

[To solution](#)

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() { // shooter function
    alert( i ); // should show its number
  };
  shooters.push(shooter);
  i++;
}

return shooters;
}

let army = makeArmy();

army[0](); // the shooter number 0 shows 10
army[5](); // and number 5 also outputs 10...
// ... all shooters show 10 instead of their 0, 1, 2, 3...

```

Why all shooters show the same? Fix the code so that they work as intended.

[Open a sandbox with tests.](#) ↗

[To solution](#)

The old "var"

In the very first chapter about [variables](#), we mentioned three ways of variable declaration:

1. `let`
2. `const`
3. `var`

`let` and `const` behave exactly the same way in terms of Lexical Environments.

But `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 meeting such scripts you may even skip this chapter or postpone it, but then there's a chance that it bites you later.

From the first sight, `var` behaves similar to `let`. That is, declares a variable:

```

function sayHi() {
  var phrase = "Hello"; // local variable, "var" instead of "let"

  alert(phrase); // Hello
}

sayHi();

alert(phrase); // Error, phrase is not defined

```

...But here are the differences.

“var” has no block scope

`var` variables are either function-wide or global, 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
```

If we used `let test` on the 2nd line, then it wouldn't be visible to `alert`. But `var` ignores code blocks, so we've got a global `test`.

The same thing for loops: `var` cannot be block- or loop-local:

```
for (var i = 0; i < 10; i++) {  
    // ...  
}  
  
alert(i); // 10, "i" 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); // Error: 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 reminiscence of that.

“var” are processed at the function start

`var` 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;  
}
```

...Is technically the same as this (moved `var phrase` above):

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

...Or even as this (remember, code blocks are ignored):

```
function sayHi() {  
    phrase = "Hello"; // (*)  
  
    if (false) {  
        var phrase;  
    }  
  
    alert(phrase);  
}
```

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 better to demonstrate with an example, like this:

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

The line `var phrase = "Hello"` has two actions in it:

1. Variable declaration `var`

2. 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`.

Summary

There are two main differences of `var`:

1. Variables have no block scope, they are visible minimum at the function level.
2. Variable declarations are processed at function start.

There's one more minor difference related to the global object, we'll cover that in the next chapter.

These differences are actually a bad thing most of the time. First, we can't create block-local variables. And hoisting just creates more space for errors. So, for new scripts `var` is used exceptionally rarely.

Global object

When JavaScript was created, there was an idea of a “global object” that provides all global variables and functions. It was planned that multiple in-browser scripts would use that single global object and share variables through it.

Since then, JavaScript greatly evolved, and that idea of linking code through global variables became much less appealing. In modern JavaScript, the concept of modules took its place.

But the global object still remains in the specification.

In a browser it is named “window”, for Node.JS it is “global”, for other environments it may have another name.

It does two things:

- Provides access to built-in functions and values, defined by the specification and the environment. For instance, we can call `alert` directly or as a method of `window`:

```
alert("Hello");

// the same as
window.alert("Hello");
```

The same applies to other built-ins. E.g. we can use `window.Array` instead of `Array`.

- Provides access to global Function Declarations and `var` variables. We can read and write them using its properties, for instance:

```
var phrase = "Hello";

function sayHi() {
  alert(phrase);
}

// can read from window
alert( window.phrase ); // Hello (global var)
alert( window.sayHi ); // function (global function declaration)

// can write to window (creates a new global variable)
window.test = 5;

alert(test); // 5
```

...But the global object does not have variables declared with `let/const`!

```
let user = "John";
alert(user); // John

alert(window.user); // undefined, don't have let
alert("user" in window); // false
```

i The global object is not a global Environment Record

In versions of ECMAScript prior to ES-2015, there were no `let/const` variables, only `var`. And global object was used as a global Environment Record (wordings were a bit different, but that's the gist).

But starting from ES-2015, these entities are split apart. There's a global Lexical Environment with its Environment Record. And there's a global object that provides *some* of the global variables.

As a practical difference, global `let/const` variables are definitely properties of the global Environment Record, but they do not exist in the global object.

Naturally, that's because the idea of a global object as a way to access "all global things" comes from ancient times. Nowadays is not considered to be a good thing. Modern language features like `let/const` do not make friends with it, but old ones are still compatible.

Uses of "window"

In server-side environments like Node.JS, the `global` object is used exceptionally rarely. Probably it would be fair to say "never".

In-browser `window` is sometimes used though.

Usually, it's not a good idea to use it, but here are some examples you can meet.

1. To access exactly the global variable if the function has the local one with the same name.

```
var user = "Global";  
  
function sayHi() {  
    var user = "Local";  
  
    alert(window.user); // Global  
}  
  
sayHi();
```

Such use is a workaround. Would be better to name variables differently, that won't require use to write the code it this way. And please note "`var`" before `user`. The trick doesn't work with `let` variables.

2. To check if a certain global variable or a builtin exists.

For instance, we want to check whether a global function `XMLHttpRequest` exists.

We can't write `if (XMLHttpRequest)`, because if there's no `XMLHttpRequest`, there will be an error (variable not defined).

But we can read it from `window.XMLHttpRequest`:

```
if (window.XMLHttpRequest) {  
    alert('XMLHttpRequest exists!')  
}
```

If there is no such global function then `window.XMLHttpRequest` is just a non-existing object property. That's `undefined`, no error, so it works.

We can also write the test without `window`:

```
if (typeof XMLHttpRequest == 'function') {  
    /* is there a function XMLHttpRequest? */  
}
```

This doesn't use `window`, but is (theoretically) less reliable, because `typeof` may use a local `XMLHttpRequest`, and we want the global one.

3. To take the variable from the right window. That's probably the most valid use case.

A browser may open multiple windows and tabs. A window may also embed another one in `<iframe>`. Every browser window has its own `window` object and global variables. JavaScript allows windows that come from the same site (same protocol, host, port) to access variables from each other.

That use is a little bit beyond our scope for now, but it looks like:

```
<iframe src="/" id="iframe"></iframe>  
  
<script>  
    alert( innerWidth ); // get innerWidth property of the current window (browser only)  
    alert( Array ); // get Array of the current window (javascript core builtin)  
  
    // when the iframe loads...  
    iframe.onload = function() {  
        // get width of the iframe window  
        alert( iframe.contentWindow.innerWidth );  
        // get the builtin Array from the iframe window  
        alert( iframe.contentWindow.Array );  
    };  
</script>
```

Here, first two alerts use the current window, and the latter two take variables from `iframe` window. Can be any variables if `iframe` originates from the same protocol/host/port.

“this” and global object

Sometimes, the value of `this` is exactly the global object. That's rarely used, but some scripts rely on that.

1. In the browser, the value of `this` in the global area is `window`:

```
// outside of functions
alert( this === window ); // true
```

Other, non-browser environments, may use another value for `this` in such cases.

2. When a function with `this` is called in non-strict mode, it gets the global object as `this`:

```
// not in strict mode (!)
function f() {
  alert(this); // [object Window]
}

f(); // called without an object
```

By specification, `this` in this case must be the global object, even in non-browser environments like Node.JS. That's for compatibility with old scripts, in strict mode `this` would be `undefined`.

Function object, NFE

As we already know, functions in JavaScript are values.

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” property

Function objects contain a few useable properties.

For instance, a function’s name is accessible as the “name” property:

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

alert(sayHi.name); // sayHi
```

What’s more funny, the name-assigning logic is smart. It also assigns the correct name to functions that are used in assignments:

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

alert(sayHi.name); // sayHi (works!)
```

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.

The idea is that we have a simple, no-arguments handler syntax for positive cases (most frequent variant), but are able to provide universal handlers as well.

To call `handlers` the right way, we examine the `length` property:

```
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 never use function properties and vice versa. These are just parallel words.

Function properties can replace closures sometimes. For instance, we can rewrite the counter function example from the chapter [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;
```

```
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 Expression

Named 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`:

1. It allows the function to reference itself internally.
2. 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 the value of `sayHi` may change. The function may go to another variable, and 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”, how the function can call itself internally.

There's no such thing for Function Declaration

The “internal name” feature described here is only available for Function Expressions, not to Function Declarations. For Function Declarations, there’s just no syntax possibility to add a one more “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. Exists not only when given in the function definition, but also for assignments and object properties.
- `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 (see the [docs](#) when you want 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.

Tasks

Set and decrease for counter

importance: 5

Modify 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 `count` to `value`.
- `counter.decrease(value)` should decrease the `count` 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.](#) ↗

[To solution](#)

Sum with an arbitrary amount of brackets

importance: 2

Write 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.

[To solution](#)

The "new Function" syntax

There's one more way to create a function. It's rarely used, but sometimes there's no alternative.

Syntax

The syntax for creating a function:

```
let func = new Function ([arg1[, arg2[, ...argN]]], functionBody)
```

In other words, function parameters (or, more precisely, names for them) go first, and the body is last. All arguments are strings.

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
```

If there are no arguments, then there's only a single argument, 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. The need for that usually arises at advanced stages of development.

Closure

Usually, a function remembers where it was born in the special property `[[Environment]]`. It references the Lexical Environment from where it's created.

But when a function is created using `new Function`, its `[[Environment]]` references not the current Lexical Environment, but instead the global one.

```
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.

Perhaps we want it to be able to access outer local 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 `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.

But, if `new Function` could access outer variables, then it would be unable to find `userName`, since this is passed in as a string *after* the code is minified.

Even if we could access outer lexical environment in `new Function`, we would have problems with minifiers.

The “special feature” of `new Function` saves us from mistakes.

And it enforces better code. If we need to pass something to a function created by `new Function`, we should pass it explicitly as an argument.

Our “sum” function actually does that right:

```
let sum = new Function('a', 'b', 'return a + b');

let a = 1, b = 2;

// outer values are passed as arguments
alert( sum(a, b) ); // 3
```

Summary

The syntax:

```
let func = new Function(arg1, arg2, ..., body);
```

For historical reasons, arguments can also be given as a comma-separated list.

These three 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 saves us from errors. Passing parameters explicitly is a much better method architecturally and causes no problems with minifiers.

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 to run a function once after the interval of time.
- `setInterval` allows to run a function regularly with the interval between the runs.

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).

arg1, arg2 ...

Arguments for the function (not supported in IE9-)

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 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 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.

Cancelling with `clearTimeout`

A 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 HTML5 standard.

setInterval

The `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);
```

Modal windows freeze time in Chrome/Opera/Safari

In browsers IE and Firefox the internal timer continues “ticking” while showing `alert/confirm/prompt`, but in Chrome, Opera and Safari the internal timer becomes “frozen”.

So if you run the code above and don’t dismiss the `alert` window for some time, then in Firefox/IE next `alert` will be shown immediately as you do it (2 seconds passed from the previous invocation), and in Chrome/Opera/Safari – after 2 more seconds (timer did not tick during the `alert`).

Recursive setTimeout

There are two ways of running something regularly.

One is `setInterval`. The other one is a recursive `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 recursive `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 we regularly have CPU-hungry tasks, then we can measure the time taken by the execution and plan the next call sooner or later.

Recursive `setTimeout` guarantees a delay between the executions, `setInterval` – does not.

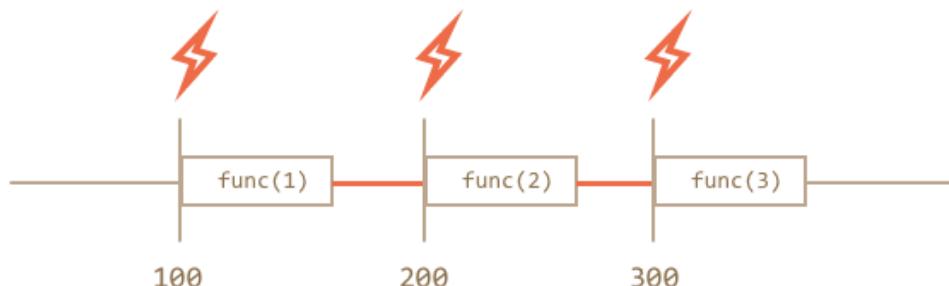
Let's compare two code fragments. The first one uses `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i);
}, 100);
```

The second one uses recursive `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!

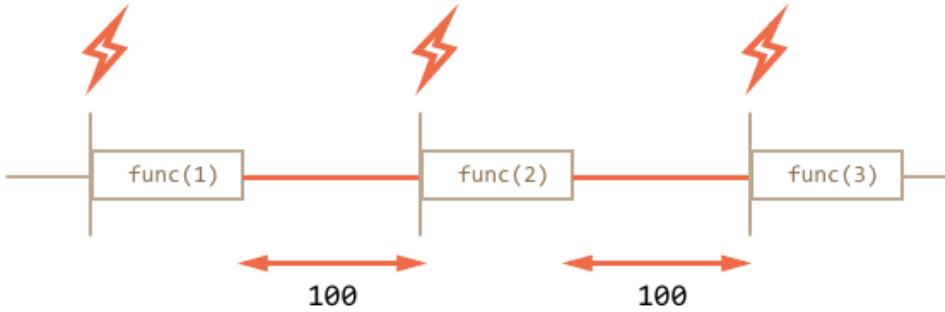
That's normal, because the time taken by `func`'s execution "consumes" a part of the interval.

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 recursive `setTimeout`:



The recursive `setTimeout` guarantees the fixed delay (here 100ms).

That's because a new call is planned at the end of the previous one.

i Garbage collection

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.

setTimeout(...,0)

There's a special use case: `setTimeout(func, 0)`.

This schedules the execution of `func` as soon as possible. But scheduler will invoke it only after the current code is complete.

So the function is scheduled to run "right after" the current code. In other words, *asynchronously*.

For instance, this outputs "Hello", then immediately "World":

```
setTimeout(() => alert("World"), 0);
alert("Hello");
```

The first line "puts the call into calendar after 0ms". But the scheduler will only "check the calendar" after the current code is complete, so "Hello" is first, and "World" – after it.

Splitting CPU-hungry tasks

There's a trick to split CPU-hungry tasks using `setTimeout`.

For instance, a syntax-highlighting script (used to colorize code examples on this page) is quite CPU-heavy. To highlight the code, it performs the analysis, creates many colored elements, adds them to the document – for a big text that takes a lot. It may even cause the browser to “hang”, which is unacceptable.

So we can split the long text into pieces. First 100 lines, then plan another 100 lines using `setTimeout(..., 0)`, and so on.

For clarity, let's take a simpler example for consideration. We have a function to count from `1` to `10000000000`.

If you run it, the CPU will hang. For server-side JS that's clearly noticeable, and if you are running it in-browser, then try to click other buttons on the page – you'll see that whole JavaScript actually is paused, no other actions work until it finishes.

```
let i = 0;

let start = Date.now();

function count() {

    // do a heavy job
    for (let j = 0; j < 1e9; j++) {
        i++;
    }

    alert("Done in " + (Date.now() - start) + 'ms');
}

count();
```

The browser may even show “the script takes too long” warning (but hopefully it won't, because the number is not very big).

Let's split the job using the nested `setTimeout`:

```
let i = 0;

let start = Date.now();

function count() {

    // do a piece of the heavy job (*)
    do {
        i++;
    } while (i % 1e6 != 0);

    if (i == 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count, 0); // schedule the new call (**)
    }
}
```

```
count();
```

Now the browser UI is fully functional during the “counting” process.

We do a part of the job (*) :

1. First run: `i=1...1000000`.
2. Second run: `i=1000001..2000000`.
3. ...and so on, the `while` checks if `i` is evenly divided by `1000000`.

Then the next call is scheduled in (**) if we’re not done yet.

Pauses between `count` executions provide just enough “breath” for the JavaScript engine to do something else, to react to other user actions.

The notable thing is that both variants – with and without splitting the job by `setTimeout` – are comparable in speed. There’s no much difference in the overall counting time.

To make them closer, let’s make an improvement.

We’ll move the scheduling in the beginning of the `count()`:

```
let i = 0;

let start = Date.now();

function count() {

    // move the scheduling at the beginning
    if (i < 1e9 - 1e6) {
        setTimeout(count, 0); // schedule the new call
    }

    do {
        i++;
    } while (i % 1e6 != 0);

    if (i == 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    }
}

count();
```

Now when we start to `count()` and know that we’ll need to `count()` more, we schedule that immediately, before doing the job.

If you run it, it’s easy to notice that it takes significantly less time.

Minimal delay of nested timers in-browser

In the browser, there's a limitation of how often nested timers can run. The [HTML5 standard](#) ↗ says: "after five nested timers, the interval is forced to be at least four milliseconds".

Let's demonstrate what it means with the example below. The `setTimeout` call in it re-schedules itself after `0ms`. 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, 0); // else re-schedule
}, 0);

// 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 the delay comes into play and we see `9, 15, 20, 24...`.

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 [process.nextTick](#) ↗ and [setImmediate](#) ↗ for Node.js. So the notion is browser-specific only.

Allowing the browser to render

Another benefit for in-browser scripts is that they can show a progress bar or something to the user. That's because the browser usually does all "repainting" after the script is complete.

So if we do a single huge function then even if it changes something, the changes are not reflected in the document till it finishes.

Here's the demo:

```
<div id="progress"></div>

<script>
  let i = 0;

  function count() {
    for (let j = 0; j < 1e6; j++) {
      i++;
      // put the current i into the <div>
      // (we'll talk more about innerHTML in the specific chapter, should be obvious here)
      progress.innerHTML = i;
    }
  }

```

```
    count();
</script>
```

If you run it, the changes to `i` will show up after the whole count finishes.

And if we use `setTimeout` to split it into pieces then changes are applied in-between the runs, so this looks better:

```
<div id="progress"></div>

<script>
  let i = 0;

  function count() {

    // do a piece of the heavy job (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);

    if (i < 1e9) {
      setTimeout(count, 0);
    }
  }

  count();
</script>
```

Now the `<div>` shows increasing values of `i`.

Summary

- Methods `setInterval(func, delay, ...args)` and `setTimeout(func, delay, ...args)` allow to run the `func` regularly/once after `delay` milliseconds.
- To cancel the execution, we should call `clearInterval/clearTimeout` with the value returned by `setInterval/setTimeout`.
- Nested `setTimeout` calls is a more flexible alternative to `setInterval`. Also they can guarantee the minimal time *between* the executions.
- Zero-timeout scheduling `setTimeout(..., 0)` is used to schedule the call “as soon as possible, but after the current code is complete”.

Some use cases of `setTimeout(..., 0)`:

- To split CPU-hungry tasks into pieces, so that the script doesn’t “hang”
- To let the browser do something else while the process is going on (paint the progress bar).

Please note that all scheduling methods do not guarantee the exact delay. We should not rely on that in the scheduled code.

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.

All that may increase the minimal timer resolution (the minimal delay) to 300ms or even 1000ms depending on the browser and settings.

Tasks

Output every second

importance: 5

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.

1. Using `setInterval`.
2. Using recursive `setTimeout`.

[To solution](#)

Rewrite `setTimeout` with `setInterval`

importance: 4

Here's the function that uses nested `setTimeout` to split a job into pieces.

Rewrite it to `setInterval`:

```
let i = 0;

let start = Date.now();

function count() {

    if (i == 1000000000) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count, 0);
    }

    // a piece of heavy job
    for(let j = 0; j < 1000000; j++) {
        i++;
    }
}
```

```
}

count();
```

[To solution](#)

What will setTimeout show?

importance: 5

In 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?

1. After the loop.
2. Before the loop.
3. 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++;
}
```

[To solution](#)

Decorators and forwarding, call/apply

JavaScript 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 caching

Let'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 for different `x` to avoid spending extra-time on recalculations.

But instead of adding that functionality into `slow()` we'll create a wrapper. 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 the result is in the map
      return cache.get(x); // return 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
alert( "Again: " + slow(1) ); // the same

alert( slow(2) ); // slow(2) is cached
alert( "Again: " + slow(2) ); // the same as the previous line

```

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.

Now let's get into details of how it works.

The result of `cachingDecorator(func)` is a “wrapper”: `function(x)` that “wraps” the call of `func(x)` into caching logic:

```

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;
  };
}

```

A red curly brace is drawn around the innermost function definition, which contains the line `let result = func(x);`. The word "wrapper" is written in orange above the brace, and the phrase "around the function" is written in orange below it, indicating that the outer function wraps the inner one.

As we can see, the wrapper returns the result of `func(x)` "as is". 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 were any).
- We can combine multiple decorators if needed (other decorators will follow).

Using “`func.call`” for the context

The 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) {
        // actually, there can be a 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 ); // this = John
sayHi.call( admin ); // this = 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:

1. After the decoration `worker.slow` is now the wrapper `function (x) { ... }`.
2. So when `worker.slow(2)` is executed, the wrapper gets `2` as an argument and `this=worker` (it's the object before dot).
3. 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 with “`func.apply`”

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);

```

We have two tasks to solve here.

First is how to use both arguments `min` and `max` for the key in `cache` map. 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:

1. Implement a new (or use a third-party) map-like data structure that is more versatile and allows multi-keys.
2. 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)`.
3. 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.

The second task to solve is how to pass many arguments to `func`. Currently, the wrapper `function(x)` assumes a single argument, and `func.call(this, x)` passes it.

Here we can use another built-in method `func.apply ↗`.

The syntax is:

```
func.apply(context, args)
```

It runs the `func` setting `this=context` and using an array-like object `args` as the list of arguments.

For instance, these two calls are almost the same:

```

func(1, 2, 3);
func.apply(context, [1, 2, 3])

```

Both run `func` giving it arguments `1, 2, 3`. But `apply` also sets `this=context`.

For instance, here `say` is called with `this=user` and `messageData` as a list of arguments:

```

function say(time, phrase) {
  alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };

let messageData = ['10:00', 'Hello']; // become time and phrase

// user becomes this, messageData is passed as a list of arguments (time, phrase)
say.apply(user, messageData); // [10:00] John: Hello (this=user)

```

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.

We already know the spread operator `...` from the chapter [Rest parameters and spread operator](#) that can pass an array (or any iterable) as a list of arguments. So if we use it with `call`, we can achieve almost the same as `apply`.

These two calls are almost equivalent:

```

let args = [1, 2, 3];

func.call(context, ...args); // pass an array as list with spread operator
func.apply(context, args); // is same as using apply

```

If we look more closely, there's a minor difference between such uses of `call` and `apply`.

- The spread operator `...` allows to pass *iterable* `args` as the list to `call`.
- The `apply` accepts only *array-like* `args`.

So, these calls complement each other. Where we expect an iterable, `call` works, where we expect an array-like, `apply` works.

And if `args` is both iterable and array-like, like a real array, then we technically could use any of them, but `apply` will probably be faster, because it's a single operation. Most JavaScript engines internally optimize it better than a pair `call + spread`.

One of the most important uses of `apply` is passing the call to another function, like this:

```

let wrapper = function() {
  return anotherFunction.apply(this, arguments);
};

```

That's called *call forwarding*. The `wrapper` passes everything it gets: the context `this` and arguments to `anotherFunction` and returns back its result.

When an external code calls such `wrapper`, it is indistinguishable from the call of the original function.

Now let's bake it all into the 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.apply(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 the wrapper operates with any number of arguments.

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.apply` to pass both the context and all arguments the wrapper got (no matter how many) to the original function.

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":

1. Let `glue` be the first argument or, if no arguments, then a comma `", "`.
2. Let `result` be an empty string.
3. Append `this[0]` to `result`.
4. Append `glue` and `this[1]`.
5. Append `glue` and `this[2]`.
6. ...Do so until `this.length` items are glued.
7. 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`.

Summary

Decorator is a wrapper around a function that alters its behavior. The main job is still carried out by the function.

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. Some decorators provide their own properties.

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.

✓ Tasks

Spy decorator

importance: 5

Create 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, it's advanced form is `sinon.spy` in [Sinon.JS ↗](#) library.

[Open a sandbox with tests. ↗](#)

[To solution](#)

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. ↗](#)

[To solution](#)

Debounce decorator

importance: 5

The result of `debounce(f, ms)` decorator should be a wrapper that passes the call to `f` at maximum once per `ms` milliseconds.

In other words, when we call a "debounced" function, it guarantees that all other future in the closest `ms` milliseconds will be ignored.

For instance:

```
let f = debounce(alert, 1000);

f(1); // runs immediately
f(2); // ignored

setTimeout( () => f(3), 100); // ignored ( only 100 ms passed )
setTimeout( () => f(4), 1100); // runs
setTimeout( () => f(5), 1500); // ignored (less than 1000 ms from the last run)
```

In practice `debounce` is useful for functions that retrieve/update something when we know that nothing new can be done in such a short period of time, so it's better not to waste resources.

[Open a sandbox with tests.](#) ↗

[To solution](#)

Throttle decorator

importance: 5

Create a “throttling” decorator `throttle(f, ms)` – that returns a wrapper, passing the call to `f` at maximum once per `ms` milliseconds. Those calls that fall into the “cooldown” period, are ignored.

The difference with `debounce` – if an ignored call is the last during the cooldown, then it executes at the end of the delay.

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 browser we can setup a function to run at every mouse micro-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).

The tracking function should update some information on the web-page.

Updating function `update()` is too heavy to do it on every micro-movement. There is also no sense in making it more often than once per 100ms.

So we'll assign `throttle(update, 100)` as the function to run on each mouse move instead of the original `update()`. The decorator will be called often, but `update()` will be called at maximum once per 100ms.

Visually, it will look like this:

1. For the first mouse movement the decorated variant passes the call to `update`. That's important, the user sees our reaction to their move immediately.

2. Then as the mouse moves on, until `100ms` nothing happens. The decorated variant ignores calls.
3. At the end of `100ms` – one more `update` happens with the last coordinates.
4. Then, finally, the mouse stops somewhere. The decorated variant waits until `100ms` expire and then runs `update` runs with last coordinates. So, perhaps the most 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.](#) ↗

[To solution](#)

Function binding

When using `setTimeout` with object methods or passing object methods along, there's a known problem: "losing `this`".

Suddenly, `this` just stops working right. The situation is typical for novice developers, but happens with experienced ones as well.

Losing “this”

We already know that in JavaScript it's easy to lose `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 as we’ll see, 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);

// ...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: bind

Functions provide a built-in method `bind ↗` that allows to fix `this`.

The basic syntax is:

```
// more complex syntax will be 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); // (*)

sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!
```

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(obj)` ↗ in lodash.

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` in an object method, so that we can pass it somewhere. For example, to `setTimeout`. There are more reasons to `bind` in the modern development, we'll meet them later.

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();
```

[To solution](#)

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();
```

[To solution](#)

Function property after bind

importance: 5

There's a value in the property of a function. Will it change after `bind`? Why, elaborate?

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

sayHi.test = 5;

let bound = sayHi.bind({
  name: "John"
});

alert( bound.test ); // what will be the output? why?
```

[To solution](#)

Ask losing this

importance: 5

The 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);
```

[To solution](#)

Currying and partials

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:

```
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 here 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:

```

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?

Here our benefit is that we created an independent function with a readable name (`double`, `triple`). We can use it and don't write the first argument of every time, cause 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 context

What if we'd like to fix some arguments, but not bind `this`?

The native `bind` does not allow that. We can't just omit the context and jump to arguments.

Fortunately, a `partial` function 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 that says something now by fixing the first argument
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 operator, right?

Also there's a ready `_partial ↗` implementation from lodash library.

Currying

Sometimes people mix up partial function application mentioned above with another thing named "currying". That's another interesting technique of working with functions that we just have to mention here.

Currying ↗ is translating a function from callable as `f(a, b, c)` into callable as `f(a)(b)(c)`.

Let's make `curry` function that performs currying for binary functions. In other words, it translates `f(a, b)` into `f(a)(b)`:

```
function curry(func) {
  return function(a) {
    return function(b) {
      return func(a, b);
    };
  };
}

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

let carriedSum = curry(sum);

alert( carriedSum(1)(2) ); // 3
```

As you can see, the implementation is a series of wrappers.

- The result of `curry(func)` is a wrapper `function(a)`.
- When it is called like `sum(1)`, the argument is saved in the Lexical Environment, and a new wrapper is returned `function(b)`.
- Then `sum(1)(2)` finally calls `function(b)` providing `2`, and it passes the call to the original multi-argument `sum`.

More advanced implementations of currying like `_curry ↗` from lodash library do something more sophisticated. They return a wrapper that allows a function to be called normally when all arguments are supplied or returns a partial otherwise.

```
function curry(f) {
  return function(...args) {
    // if args.length == f.length (as many arguments as f has),
```

```
// then pass the call to f
// otherwise return a partial function that fixes args as first arguments
};

}
```

Currying? What for?

Advanced currying allows both to keep the function callable normally and to get partials easily. To understand the benefits we definitely 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 also have many other useful features like: sending it over the network or filtering:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}]:${date.getMinutes()}] [${importance}] ${message}`);
}
```

Let's curry it!

```
log = _.curry(log);
```

After that `log` still works the normal way:

```
log(new Date(), "DEBUG", "some debug");
```

...But also can be called in the curried form:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Let's get a convenience function for today's logs:

```
// todayLog will be the partial of log with fixed first argument
let todayLog = log(new Date());

// use it
todayLog("INFO", "message"); // [HH:mm] INFO message
```

And now a convenience function for today's debug messages:

```
let todayDebug = todayLog("DEBUG");

todayDebug("message"); // [HH:mm] DEBUG message
```

So:

1. We didn't lose anything after currying: `log` is still callable normally.
2. We were able to generate partial functions that are convenient in many cases.

Advanced curry implementation

In case you're interested, here's the “advanced” curry implementation that we could use above.

```
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));  
      }  
    }  
  };  
}  
  
function sum(a, b, c) {  
  return a + b + c;  
}  
  
let curriedSum = curry(sum);  
  
// still callable normally  
alert( curriedSum(1, 2, 3) ); // 6  
  
// get the partial with curried(1) and call it with 2 other arguments  
alert( curriedSum(1)(2,3) ); // 6  
  
// full curried form  
alert( curriedSum(1)(2)(3) ); // 6
```

The new `curry` may look complicated, but it's actually pretty easy to understand.

The result of `curry(func)` 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 pass(...args2) { // (2)  
      return curried.apply(this, args.concat(args2));  
    }  
  }  
};
```

When we run it, there are two branches:

1. Call now: if passed `args` count is the same as the original function has in its definition (`func.length`) or longer, then just pass the call to it.
2. Get a partial: otherwise, `func` is not called yet. Instead, another wrapper `pass` is returned, that will re-apply `curried` providing previous arguments together with the new ones. Then on a new call, again, we'll get either a new partial (if not enough arguments) or, finally, the result.

For instance, let's see what happens in the case of `sum(a, b, c)`. Three arguments, so `sum.length = 3`.

For the call `curried(1)(2)(3)`:

1. The first call `curried(1)` remembers `1` in its Lexical Environment, and returns a wrapper `pass`.
2. The wrapper `pass` is called with `(2)`: it takes previous args `(1)`, concatenates them with what it got `(2)` and calls `curried(1, 2)` with them together.
As the argument count is still less than 3, `curry` returns `pass`.
3. The wrapper `pass` is called again with `(3)`, for the next call `pass(3)` takes previous args `(1, 2)` and adds `3` to them, making the call `curried(1, 2, 3)` – there are `3` arguments at last, they are given to the original function.

If that's still not obvious, just trace the calls sequence in your mind or on the paper.

Fixed-length functions only

The currying requires the function to have a known fixed number of arguments.

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.

Summary

- When we fix some arguments of an existing function, the resulting (less universal) function is called a *partial*. We can use `bind` to get a partial, but there are other ways also.

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.

- *Currying* 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 arguments count is not enough.

Currying is great when we want easy partials. As we've seen in the logging example: the universal function `log(date, importance, message)` after currying gives us partials when called with one argument like `log(date)` or two arguments `log(date, importance)`.

Tasks

Partial application for login

importance: 5

The task is a little more complex variant of [Ask losing this](#).

The `user` object was modified. Now instead of two functions `loginOk/loginFail`, it has a single function `user.login(true/false)`.

What to 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.

[To solution](#)

Arrow functions revisited

Let's revisit arrow functions.

Arrow functions are not just a “shorthand” for writing small stuff.

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.

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`.

ℹ️ 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.

Summary

Arrow 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. Will be in the chapter [Class inheritance, super](#)).

That's because they are meant for short pieces of code that do not have their own "context", but rather works in the current one. And they really shine in that use case.

Objects, classes, inheritance

In this section we return to objects and learn them even more in-depth.

Property flags and descriptors

As we know, objects can store properties.

Till now, a property was a simple "key-value" pair to us. But an object property is actually a more complex and tunable thing.

Property flags

Object properties, besides a `value`, have three special attributes (so-called "flags"):

- `writable` – if `true`, 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 property to work on.

descriptor

Property descriptor 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.

Read-only

Let’s make `user.name` read-only 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.

Here’s the same operation, but for the case when a property doesn’t exist:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "Pete",
  // for new properties need to explicitly list what's true
  enumerable: true,
  configurable: true
});

alert(user.name); // Pete
user.name = "Alice"; // 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 `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 `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 not be deleted or altered with `defineProperty`.

For instance, `Math.PI` is read-only, 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  
  
// delete Math.PI won't work either
```

Making a property non-configurable is a one-way road. We cannot change it back, because `defineProperty` doesn't work on non-configurable properties.

Here we are making `user.name` a “forever sealed” constant:

```
let user = { };  
  
Object.defineProperty(user, "name", {  
    value: "John",  
    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  
//   defineProperty(user, "name", ...)  
Object.defineProperty(user, "name", {writable: true}); // Error
```

i Errors appear only in use strict

In the non-strict mode, no errors occur when writing to read-only properties and such. But the operation still won't succeed. Flag-violating actions are just silently ignored in non-strict.

Object.defineProperties

There'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.defineProperties(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.defineProperties` it can be used as a “flags-aware” way of cloning an object:

```
let clone = Object.defineProperties({}, 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.defineProperties` is preferred.

Another difference is that `for..in` ignores symbolic properties, but

`Object.getOwnPropertyDescriptors` returns *all* property descriptors including symbolic ones.

Sealing an object globally

Property descriptors work at the level of individual properties.

There are also methods that limit access to the *whole* object:

[Object.preventExtensions\(obj\)](#)

Forbids to add properties to the object.

[Object.seal\(obj\)](#)

Forbids to add/remove properties, sets for all existing properties `configurable: false`.

[Object.freeze\(obj\)](#)

Forbids to add/remove/change properties, sets for all existing properties `configurable: false, writable: false`.

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.

Property getters and setters

There are two kinds of properties.

The first kind is *data properties*. We already know how to work with them. Actually, all properties that we've been using till now were data properties.

The second type of properties is something new. It's *accessor properties*. They are essentially functions that work 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 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’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
```

Now we have a “virtual” property. It is readable and writable, but in fact does not exist.

Accessor properties are only accessible with `get/set`

A property can either be a “data property” or an “accessor property”, but not both.

Once a property is defined with `get prop()` or `set prop()`, it’s an accessor property. So there must be a getter to read it, and must be a setter if we want to assign it.

Sometimes it’s normal that there’s only a setter or only a getter. But the property won’t be readable or writable in that case.

Accessor descriptors

Descriptors for accessor properties are different – as compared with data properties.

For accessor properties, there is no `value` and `writable`, but instead there are `get` and `set` functions.

So 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 once again that a property can be either an accessor or a data property, 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 them.

For instance, if we want to forbid too short names for `user`, we can store `name` in a special property `_name`. And filter assignments in the setter:

```
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...
```

Technically, the external code may still access the name directly by using `user._name`. But there is a widely known agreement that properties starting with an underscore `_` are internal and should not be touched from outside the object.

Using for compatibility

One of the great ideas behind getters and setters – they allow to take control over a “normal” data property and tweak it at any moment.

For instance, 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 written by other people. And besides, `age` is a nice thing to have in `user`, right? In some places it's just what we want.

Adding a getter for `age` mitigates 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.

Prototypal inheritance

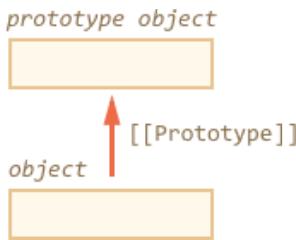
In programming, we often want to take something and extend it.

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”:



That `[[Prototype]]` has a “magical” meaning. When we want to read a property from `object`, and it’s missing, JavaScript automatically takes it from the prototype. In programming, such thing is called “prototypal inheritance”. Many cool language features and programming techniques are based on it.

The property `[[Prototype]]` is internal and hidden, but there are many ways to set it.

One of them is to use `__proto__`, like this:

```

let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

```

Please note that `__proto__` is *not the same* as `[[Prototype]]`. That’s a getter/setter for it. We’ll talk about other ways of setting it later, but for now `__proto__` will do just fine.

If we look for a property in `rabbit`, and it’s missing, JavaScript automatically takes it from `animal`.

For instance:

```

let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

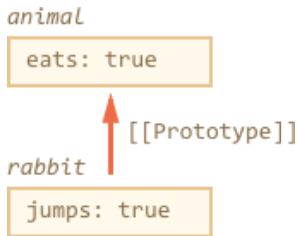
rabbit.__proto__ = animal; // (*)

// we can find both properties in rabbit now:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

Here the line `(*)` sets `animal` to be a 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 "rabit" or "rabit" prototypally inherits from "animal".

So if `animal` has a lot of useful properties and methods, then they become automatically available in `rabit`. Such properties are called "inherited".

If we have a method in `animal`, it can be called on `rabit`:

```

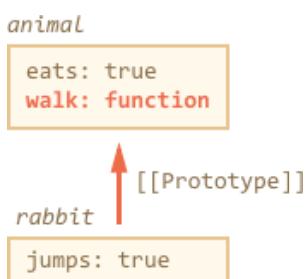
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

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)

```

animal

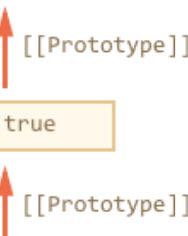
```
eats: true
walk: function
```

rabbit

```
jumps: true
```

LongEar

```
earLength: 10
```



There are actually only two limitations:

1. The references can't go in circles. JavaScript will throw an error if we try to assign `__proto__` in a circle.
2. The value of `__proto__` can be either an object or `null`. All other values (like primitives) are ignored.

Also it may be obvious, but still: there can be only one `[[Prototype]]`. An object may not inherit from two others.

Read/write rules

The prototype is only used for reading properties.

For data properties (not getters/setters) 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 */
  }
};

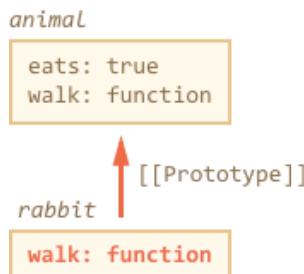
let rabbit = {
  __proto__: animal
}

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:



For getters/setters – if we read/write a property, they are looked up in the prototype and invoked.

For instance, check out `admin.fullName` property 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"; // (**)
```

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 the properties `this.name` and `this.surname` are written: `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 actually uses `admin` as `this`, not `user`.

That is actually a super-important thing, because we may have a big object with many methods and inherit from it. Then we can run its methods on inherited objects and they will modify the state of these objects, not the big one.

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:

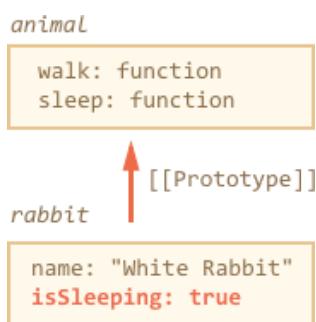
```
// animal has methods
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

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 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.

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 (there are other ways too, 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 work directly on the object, they don't use the prototype (unless the property is actually 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.

Tasks

Working with prototype

importance: 5

Here'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.

[To solution](#)

Searching algorithm

importance: 5

The task has two parts.

We have an object:

```
let head = {  
    glasses: 1
```

```
};

let table = {
  pen: 3
};

let bed = {
  sheet: 1,
  pillow: 2
};

let pockets = {
  money: 2000
};
```

1. 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`).
2. Answer the question: is it faster to get `glasses` as `pockets.glasses` or `head.glasses`? Benchmark if needed.

[To solution](#)

Where it writes?

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();
```

[To solution](#)

Why two hamsters are 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 to 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

```

[To solution](#)

F.prototype

In modern JavaScript we can set a prototype using `__proto__`, as described in the previous article. But it wasn't like that all the time.

JavaScript has had prototypal inheritance from the beginning. It was one of the core features of the language.

But in the old times, there was another (and the only) way to set it: to use a "prototype" property of the constructor function. And there are still many scripts that use it.

The “prototype” property

As we know already, `new F()` creates a new object.

When a new object is created with `new F()`, the object's `[[Prototype]]` is set to `F.prototype`.

In other words, if `F` has a `prototype` property with a value of the object type, then `new` operator uses it to set `[[Prototype]]` for the new object.

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;

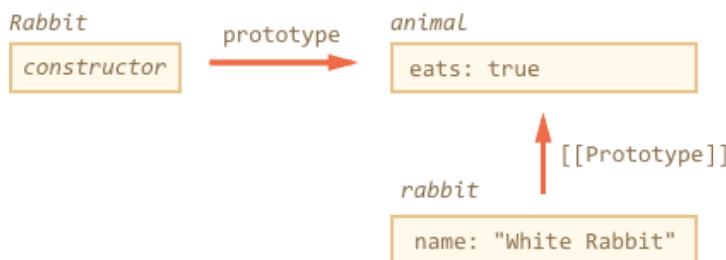
let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal

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, it's a regular property, and `[[Prototype]]` is vertical, meaning the inheritance of `rabbit` from `animal`.

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.

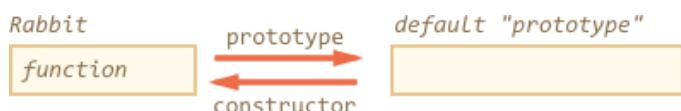
Like this:

```

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]]`:

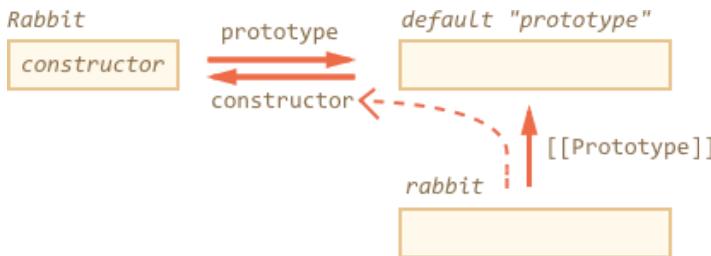
```

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 it 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 few notes to make things clear:

- The `F.prototype` property is not the same as `[[Prototype]]`. The only thing `F.prototype` does: it 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 is set to 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.

✓ Tasks

Changing "prototype"

importance: 5

In 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
```

1. We added one more string (emphasized), what `alert` shows now?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype = {};

alert( rabbit.eats ); // ?
```

2. ...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 ); // ?
```

3. Like this (replaced one line)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};
```

```
};

let rabbit = new Rabbit();

delete rabbit.eats;

alert( rabbit.eats ); // ?
```

4. The last variant:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete Rabbit.prototype.eats;

alert( rabbit.eats ); // ?
```

[To solution](#)

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.

[To solution](#)

Native prototypes

The "prototype" property is widely used by the core of JavaScript itself. All built-in constructor functions use it.

We'll see how it is for plain objects first, and then for more complex ones.

Object.prototype

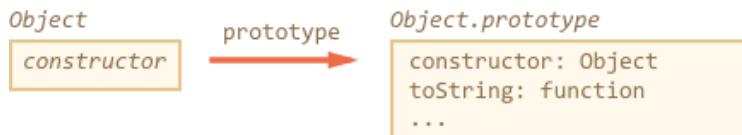
Let's say we output an empty object:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

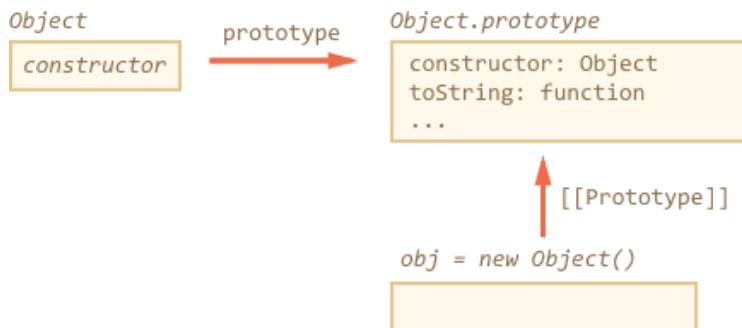
Where's the code that generates the string "[object Object]"? That's a built-in `toString` method, but where is it? The `obj` is empty!

...But the short notation `obj = {}` is the same as `obj = new Object()`, where `Object` – is a built-in object constructor function. And that function has `Object.prototype` that references a huge object with `toString` and other functions.

Like this (all that is built-in):



When `new Object()` is called (or a literal object `{...}` is created), the `[[Prototype]]` of it is set to `Object.prototype` by the rule that we've discussed in the previous chapter:



Afterwards 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
// obj.toString === obj.__proto__.toString == Object.prototype.toString
```

Please note that there is no additional `[[Prototype]]` in the chain above `Object.prototype`:

```
alert(Object.prototype.__proto__); // null
```

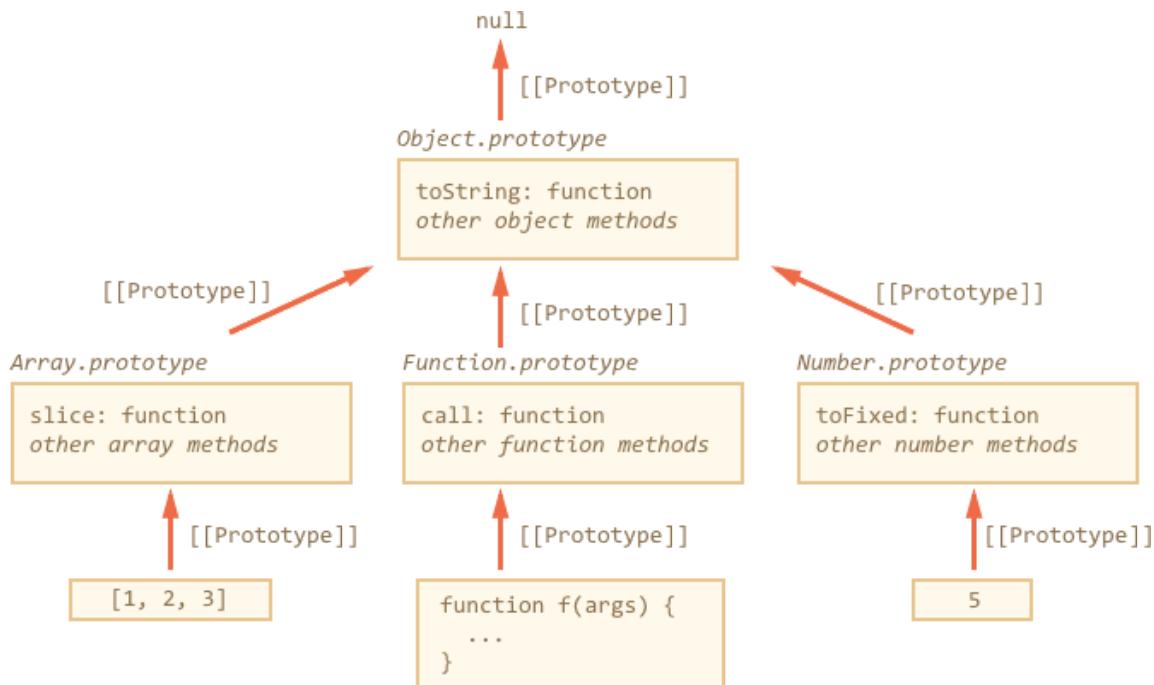
Other built-in prototypes

Other 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 the array data is written into the new object, and `Array.prototype` becomes its prototype and provides methods. That's very memory-efficient.

By specification, all built-in prototypes have `Object.prototype` on the top. Sometimes 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

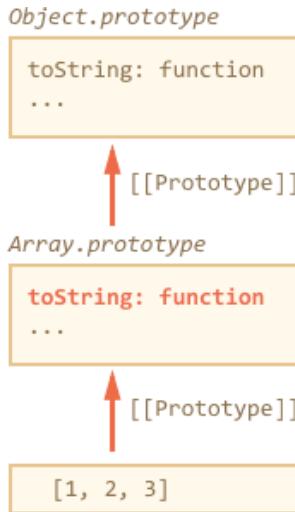
// then from Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// 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 (may need to use `console.dir` for built-in objects):

```

> console.dir([1,2,3])
  ▼ Array[3] ⓘ
    0: 1
    1: 2
    2: 3
    length: 3
  ▼ __proto__:=Array.prototype
    ► concat: function concat() { [native code] }
    ► ...
    ► unshift: function unshift() { [native code] }
  ▼ __proto__:=Object.prototype
    ► ...
    ► constructor: function Object() { [native code] }
    ► hasOwnProperty: function hasOwnProperty() { [native code] }
    ► isPrototypeOf: function isPrototypeOf() { [native code] }
    ► ...

```

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, then temporary wrapper objects are created using built-in constructors `String`, `Number`, `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 too.

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 which new built-in methods we'd like to have. And there may be a slight temptation to add them to native prototypes. But that is generally a bad idea.

Prototypes are global, so it's easy to get a conflict. If two libraries add a method `String.prototype.show`, then one of them overwrites the other one.

In modern programming, there is only one case when modifying native prototypes is approved. That's polyfills. In other words, if there's a method in JavaScript specification that is not yet supported by our JavaScript engine (or any of those that we want to support), then may 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 more complex than that,
    // throw errors for negative values of "n"
    // the full algorithm is in the specification
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Borrowing from prototypes

In the chapter [Decorators and forwarding, `call/apply`](#) we talked about method borrowing:

```

function showArgs() {
  // borrow join from array and call in the context of arguments
  alert( [].join.call(arguments, " - ") );
}

showArgs("John", "Pete", "Alice"); // John - Pete - Alice

```

Because `join` resides in `Array.prototype`, we can call it from there directly and rewrite it as:

```

function showArgs() {
  alert( Array.prototype.join.call(arguments, " - ") );
}

```

That's more efficient, because it avoids the creation of an extra array object `[]`. On the other hand, it is longer to write.

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`, `Boolean.prototype`. There are no wrapper objects only for `undefined` and `null`.
- Built-in prototypes can be modified or populated with new methods. But it's not recommended to change them. Probably the only allowable cause is when we add-in a new standard, but not yet supported by the engine JavaScript method.

Tasks

Add method "f.defer(ms)" to functions

importance: 5

Add 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

```

[To solution](#)

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.

[To solution](#)

Methods for prototypes

In this chapter we cover additional methods to work with a prototype.

There are also other ways to get/set a prototype, besides those that we already know:

- `Object.create(proto[, descriptors])` ↪ – creates an empty object with given `proto` as `[[Prototype]]` and optional property descriptors.
- `Object.getPrototypeOf(obj)` ↪ – returns the `[[Prototype]]` of `obj`.
- `Object.setPrototypeOf(obj, proto)` ↪ – sets the `[[Prototype]]` of `obj` to `proto`.

For instance:

```
let animal = {
  eats: true
};

// create a new object with animal as a prototype
let rabbit = Object.create(animal);

alert(rabbit.eats); // true
alert(Object.getPrototypeOf(rabbit) === animal); // get the prototype of rabbit

Object.setPrototypeOf(rabbit, {}); // change the prototype of rabbit to {}
```

`Object.create` 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`:

```

// fully identical shallow clone of obj
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

If we count all the ways to manage `[[Prototype]]`, there's a lot! Many ways to do the same!

Why so?

That's for historical reasons.

- The "prototype" property of a constructor function works since very ancient times.
- Later in the year 2012: `Object.create` appeared in the standard. It allowed to create objects with the given prototype, but did not allow to get/set it. So browsers implemented non-standard `__proto__` accessor that allowed to get/set a prototype at any time.
- Later in the year 2015: `Object.setPrototypeOf` and `Object.getPrototypeOf` were added to the standard. The `__proto__` was de-facto implemented everywhere, so it made its way to the Annex B of the standard, that is optional for non-browser environments.

As of now we have all these ways at our disposal.

Technically, we can get/set `[[Prototype]]` at any time. But usually we only set it once at the object creation time, and then do not modify: `rabbit` inherits from `animal`, and that is not going to change. And JavaScript engines are highly optimized to that. Changing a prototype "on-the-fly" with `Object.setPrototypeOf` or `obj.__proto__ =` is a very slow operation. But it is possible.

“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 is ignored!

That shouldn't surprise us. The `__proto__` property is special: it must be either an object or `null`, a string can not become a prototype.

But we did not 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 the prototype may indeed be changed, so the execution may go wrong in totally unexpected ways.

What's worst – 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.

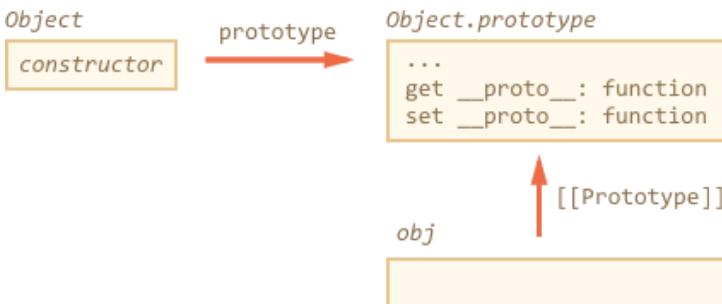
Such thing happens only with `__proto__`. All other properties are “assignable” normally.

How to evade the problem?

First, we can just switch to using `Map`, then everything's fine.

But `Object` also can serve us well here, because language creators gave a thought to that problem long ago.

The `__proto__` is not a property of an object, but an accessor property of `Object.prototype`:



So, if `obj.__proto__` is read or assigned, the corresponding getter/setter is called from its prototype, and it gets/sets `[[Prototype]]`.

As it was said in the beginning: `__proto__` is a way to access `[[Prototype]]`, it is not `[[Prototype]]` itself.

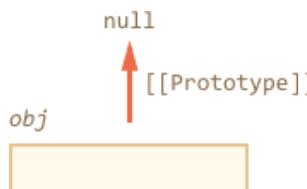
Now, if we want to use an object as an associative array, we can do it with a little trick:

```
let obj = Object.create(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 object “very plain” or “pure dictionary objects”, because they are even simpler than 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.

Please 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 = "ni hao";
chineseDictionary.bye = "zai jian";

alert(Object.keys(chineseDictionary)); // hello,bye
```

Getting all properties

There are many ways to get keys/values from an object.

We already know these ones:

- [Object.keys\(obj\)](#) / [Object.values\(obj\)](#) / [Object.entries\(obj\)](#) – returns an array of enumerable own string property names/values/key-value pairs. These methods only list *enumerable* properties, and those that have *strings* as *keys*.

If we want symbolic properties:

- [Object.getOwnPropertySymbols\(obj\)](#) – returns an array of all own symbolic property names.

If we want non-enumerable properties:

- [Object.getOwnPropertyNames\(obj\)](#) – returns an array of all own string property names.

If we want *all* properties:

- [Reflect.ownKeys\(obj\)](#) – returns an array of all own property names.

These methods are a bit different about which properties they return, but all of them operate on the object itself. Properties from the prototype are not listed.

The `for .. in` loop is different: it loops over inherited properties too.

For instance:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// only own keys
alert(Object.keys(rabbit)); // jumps

// inherited keys too
for(let prop in rabbit) alert(prop); // jumps, then eats
```

If we want to distinguish 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
};

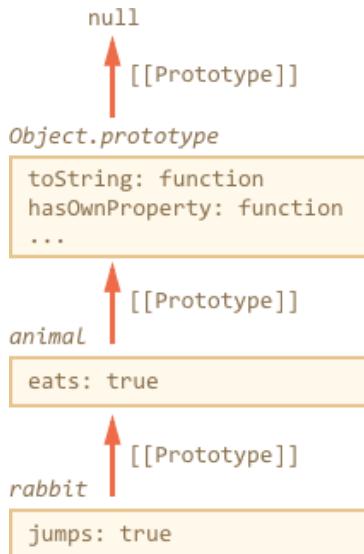
let rabbit = {
  jumps: true,
  __proto__: animal
};
```

```

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);
  alert(` ${prop}: ${isOwn}`); // jumps:true, then eats:false
}

```

Here we have the following inheritance chain: `rabbit`, then `animal`, then `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? Looking at the chain we can see that the method is provided by `Object.prototype.hasOwnProperty`. In other words, it's inherited.

...But why `hasOwnProperty` does not appear in `for..in` loop, if it lists all inherited properties? The answer is simple: it's not enumerable. Just like all other properties of `Object.prototype`. That's why they are not listed.

Summary

Here's a brief list of methods we discussed in this chapter – as a recap:

- [Object.create\(proto\[, descriptors\]\)](#) – creates an empty object with given `proto` as `[[Prototype]]` (can be `null`) and optional property descriptors.
- [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).
- [Object.keys\(obj\)](#) / [Object.values\(obj\)](#) / [Object.entries\(obj\)](#) – returns an array of enumerable own string property names/values/key-value pairs.
- [Object.getOwnPropertySymbols\(obj\)](#) – returns an array of all own symbolic property names.
- [Object.getOwnPropertyNames\(obj\)](#) – returns an array of all own string property names.
- [Reflect.ownKeys\(obj\)](#) – returns an array of all own property names.

- `obj.hasOwnProperty(key)` ↳ it returns `true` if `obj` has its own (not inherited) property named `key`.

We also made it clear that `__proto__` is a getter/setter for `[[Prototype]]` and resides in `Object.prototype`, just as other methods.

We can create an object without a prototype by `Object.create(null)`. Such objects are used as “pure dictionaries”, they have no issues with `"__proto__"` as the key.

All methods that return object properties (like `Object.keys` and others) – return “own” properties. If we want inherited ones, then we can use `for..in`.

✓ Tasks

Add `toString` to the dictionary

importance: 5

There'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__"
```

[To solution](#)

The difference beteeen calls

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();
```

[To solution](#)

Class patterns

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

“ Wikipedia

There's a special syntax construct and a keyword `class` in JavaScript. But before studying it, we should consider that the term “class” comes from the theory of object-oriented programming. The definition is cited above, and it's language-independent.

In JavaScript there are several well-known programming patterns to make classes even without using the `class` keyword. And here we'll talk about them first.

The `class` construct will be described in the next chapter, but in JavaScript it's a “syntax sugar” and an extension of one of the patterns that we'll study here.

Functional class pattern

The constructor function below can be considered a “class” according to the definition:

```
function User(name) {
  this.sayHi = function() {
    alert(name);
  };
}

let user = new User("John");
user.sayHi(); // John
```

It follows all parts of the definition:

1. It is a “program-code-template” for creating objects (callable with `new`).
2. It provides initial values for the state (`name` from parameters).
3. It provides methods (`sayHi`).

This is called *functional class pattern*.

In the functional class pattern, local variables and nested functions inside `User`, that are not assigned to `this`, are visible from inside, but not accessible by the outer code.

So we can easily add internal functions and variables, like `calcAge()` here:

```
function User(name, birthday) {
  // only visible from other methods inside User
  function calcAge() {
    return new Date().getFullYear() - birthday.getFullYear();
  }

  this.sayHi = function() {
    alert(` ${name}, age:${calcAge()}`);
  };
}

let user = new User("John", new Date(2000, 0, 1));
user.sayHi(); // John, age:17
```

In this code variables `name`, `birthday` and the function `calcAge()` are internal, *private* to the object. They are only visible from inside of it.

On the other hand, `sayHi` is the external, *public* method. The external code that creates `user` can access it.

This way we can hide internal implementation details and helper methods from the outer code. Only what's assigned to `this` becomes visible outside.

Factory class pattern

We can create a class without using `new` at all.

Like this:

```
function User(name, birthday) {
  // only visible from other methods inside User
  function calcAge() {
    return new Date().getFullYear() - birthday.getFullYear();
  }

  return {
    sayHi() {
      alert(` ${name}, age:${calcAge()}`);
    }
  };
}
```

```
let user = User("John", new Date(2000, 0, 1));
user.sayHi(); // John, age:17
```

As we can see, the function `User` returns an object with public properties and methods. The only benefit of this method is that we can omit `new`: write `let user = User(...)` instead of `let user = new User(...)`. In other aspects it's almost the same as the functional pattern.

Prototype-based classes

Prototype-based classes are the most important and generally the best. Functional and factory class patterns are rarely used in practice.

Soon you'll see why.

Here's the same class rewritten using prototypes:

```
function User(name, birthday) {
  this._name = name;
  this._birthday = birthday;
}

User.prototype._calcAge = function() {
  return new Date().getFullYear() - this._birthday.getFullYear();
};

User.prototype.sayHi = function() {
  alert(` ${this._name}, age:${this._calcAge()}`);
};

let user = new User("John", new Date(2000, 0, 1));
user.sayHi(); // John, age:17
```

The code structure:

- The constructor `User` only initializes the current object state.
- Methods are added to `User.prototype`.

As we can see, methods are lexically not inside `function User`, they do not share a common lexical environment. If we declare variables inside `function User`, then they won't be visible to methods.

So, there is a widely known agreement that internal properties and methods are prepended with an underscore `_`. Like `_name` or `_calcAge()`. Technically, that's just an agreement, the outer code still can access them. But most developers recognize the meaning of `_` and try not to touch prefixed properties and methods in the external code.

Here are the advantages over the functional pattern:

- In the functional pattern, each object has its own copy of every method. We assign a separate copy of `this.sayHi = function() { ... }` and other methods in the constructor.

- In the prototypal pattern, all methods are in `User.prototype` that is shared between all user objects. An object itself only stores the data.

So the prototypal pattern is more memory-efficient.

...But not only that. Prototypes allow us to setup the inheritance in a really efficient way. Built-in JavaScript objects all use prototypes. Also there's a special syntax construct: "class" that provides nice-looking syntax for them. And there's more, so let's go on with them.

Prototype-based inheritance for classes

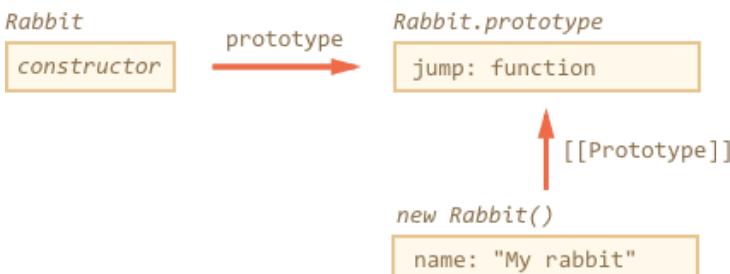
Let's say we have two prototype-based classes.

`Rabbit`:

```
function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype.jump = function() {
  alert(` ${this.name} jumps! `);
};

let rabbit = new Rabbit("My rabbit");
```

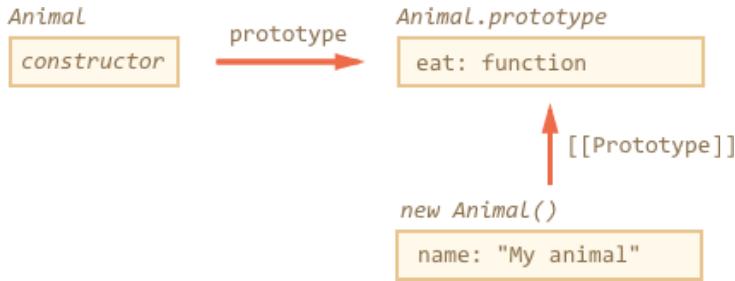


...And `Animal`:

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.eat = function() {
  alert(` ${this.name} eats. `);
};

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



Right now they are fully independent.

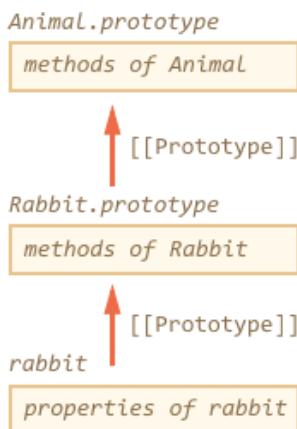
But we'd want `Rabbit` to extend `Animal`. In other words, rabbits should be based on animals, have access to methods of `Animal` and extend them with its own methods.

What does it mean in the language of prototypes?

Right now methods for `rabbit` objects are in `Rabbit.prototype`. We'd like `rabbit` to use `Animal.prototype` as a “fallback”, if the method is not found in `Rabbit.prototype`.

So the prototype chain should be `rabbit` → `Rabbit.prototype` → `Animal.prototype`.

Like this:



The code to implement that:

```

// Same Animal as before
function Animal(name) {
  this.name = name;
}

// All animals can eat, right?
Animal.prototype.eat = function() {
  alert(` ${this.name} eats.`);
};

// Same Rabbit as before
function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype.jump = function() {
  alert(` ${this.name} jumps!`);
};

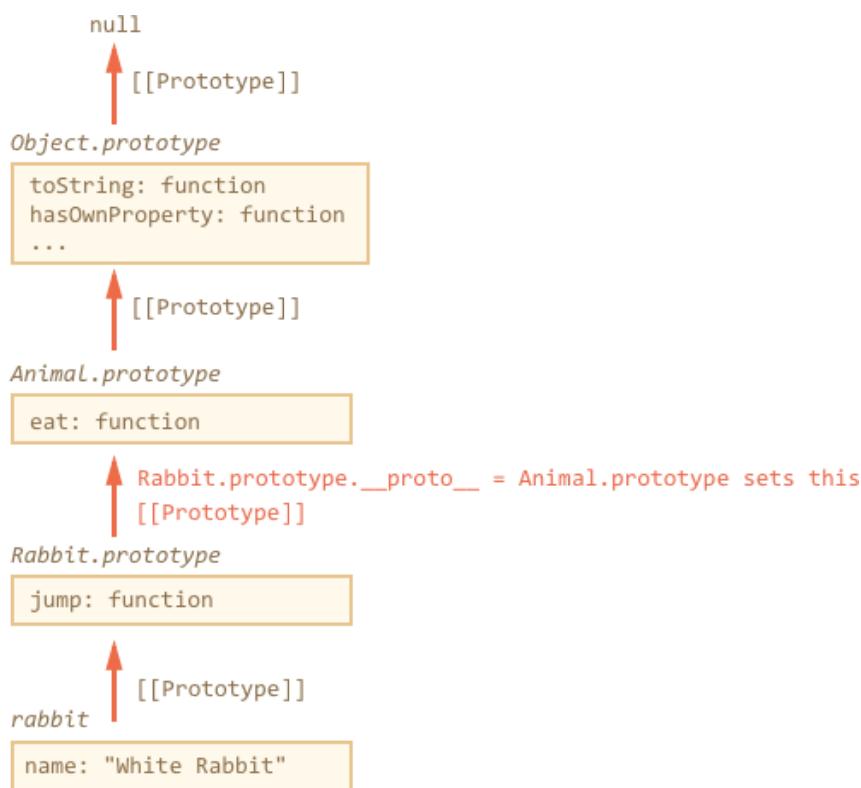
```

```
// setup the inheritance chain
Rabbit.prototype.__proto__ = Animal.prototype; // (*)

let rabbit = new Rabbit("White Rabbit");
rabbit.eat(); // rabbits can eat too
rabbit.jump();
```

The line `(*)` sets up the prototype chain. So that `rabbit` first searches methods in `Rabbit.prototype`, then `Animal.prototype`. And then, just for completeness, let's mention that if the method is not found in `Animal.prototype`, then the search continues in `Object.prototype`, because `Animal.prototype` is a regular plain object, so it inherits from it.

So here's the full picture:



Summary

The term “class” comes from the object-oriented programming. In JavaScript it usually means the functional class pattern or the prototypal pattern. The prototypal pattern is more powerful and memory-efficient, so it's recommended to stick to it.

According to the prototypal pattern:

1. Methods are stored in `Class.prototype`.
2. Prototypes inherit from each other.

In the next chapter we'll study `class` keyword and construct. It allows to write prototypal classes shorter and provides some additional benefits.

Tasks

An error in the inheritance

importance: 5

Find an error in the prototypal inheritance below.

What's wrong? What are consequences going to be?

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.walk = function() {
  alert(this.name + ' walks');
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = Animal.prototype;

Rabbit.prototype.walk = function() {
  alert(this.name + " bounces!");
};
```

[To solution](#)

Rewrite to prototypes

importance: 5

The `Clock` class is written in functional style. Rewrite it using prototypes.

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

[Open a sandbox for the task.](#) ↗

[To solution](#)

Classes

The “class” construct allows to define prototype-based classes with a clean, nice-looking syntax.

The “class” syntax

The `class` syntax is versatile, we'll start with a simple example first.

Here's a prototype-based class `User`:

```

function User(name) {
  this.name = name;
}

User.prototype.sayHi = function() {
  alert(this.name);
}

let user = new User("John");
user.sayHi();

```

...And that's the same using `class` syntax:

```

class User {

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

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

let user = new User("John");
user.sayHi();

```

It's easy to see that the two examples are alike. Just please note that methods in a class do not have a comma between them. Novice developers sometimes forget it and put a comma between class methods, and things don't work. That's not a literal object, but a class syntax.

So, what exactly does `class` do? We may think that it defines a new language-level entity, but that would be wrong.

The `class User { . . . }` here actually does two things:

1. Declares a variable `User` that references the function named "constructor".
2. Puts methods listed in the definition into `User.prototype`. Here, it includes `sayHi` and the `constructor`.

Here's the code to dig into the class and see that:

```

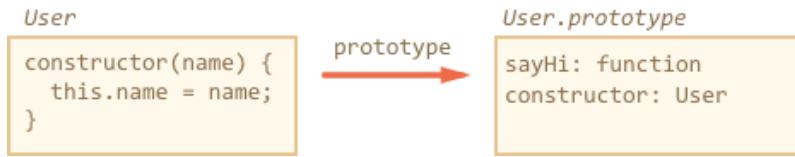
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// proof: User is the "constructor" function
alert(User === User.prototype.constructor); // true

// proof: there are two methods in its "prototype"
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

```

Here's the illustration of what `class User` creates:



So `class` is a special syntax to define a constructor together with its prototype methods.

...But not only that. There are minor tweaks here and there:

Constructors require `new`

Unlike a regular function, a class `constructor` can't be called without `new`:

```
class User {  
  constructor() {}  
}  
  
alert(typeof User); // function  
User(); // Error: Class constructor User cannot be invoked without 'new'
```

Different string output

If we output it like `alert(User)`, some engines show "class User...", while others show "function User...".

Please don't be confused: the string representation may vary, but that's still a function, there is no separate "class" entity in JavaScript language.

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 have a default `constructor() {}`

If there's no `constructor` in the `class` construct, then an empty function is generated, same as if we had written `constructor() {}`.

Classes always use strict

All code inside the class construct is automatically in strict mode.

Getters/setters

Classes may also include getters/setters. Here's an example with `user.name` implemented using them:

```
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 too short.

```

Internally, getters and setters are also created on the `User` prototype, like this:

```

Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});

```

Only methods

Unlike object literals, no `property:value` assignments are allowed inside `class`. There may be only methods and getters/setters. There is some work going on in the specification to lift that limitation, but it's not yet there.

If we really need to put a non-function value into the prototype, then we can alter `prototype` manually, like this:

```

class User {}

User.prototype.test = 5;

alert( new User().test ); // 5

```

So, technically that's possible, but we should know why we're doing it. Such properties will be shared among all objects of the class.

An “in-class” alternative is to use a getter:

```

class User {
  get test() {
    return 5;
  }
}

alert( new User().test ); // 5

```

From the external code, the usage is the same. But the getter variant is a bit slower.

Class Expression

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

Here's a class-returning function ("class factory"):

```

function makeClass(phrase) {
  // declare a class and return it
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}

let User = makeClass("Hello");

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

```

That's quite normal if we recall that `class` is just a special form of a function-with-prototype definition.

And, like Named Function Expressions, such classes also may have a name, that is visible inside that class only:

```

// "Named Class Expression" (alas, no such term, but that's what's going on)
let User = class MyClass {
  sayHi() {
    alert(MyClass); // MyClass is visible only inside the class
  }
};

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

alert(MyClass); // error, MyClass not visible outside of the class

```

Static methods

We can also assign methods to the class function, not to its "prototype". Such methods are called *static*.

An example:

```
class User {  
    static staticMethod() {  
        alert(this === User);  
    }  
}  
  
User.staticMethod(); // true
```

That actually does the same as assigning it as a function property:

```
function User() {}  
  
User.staticMethod = function() {  
    alert(this === User);  
};
```

The value of `this` inside `User.staticMethod()` is the class constructor `User` itself (the “object before dot” rule).

Usually, static methods are used to implement functions that belong to the class, but not to any particular object of it.

For instance, we have `Article` objects and need a function to compare them. The natural choice would be `Article.compare`, like this:

```
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("Mind", new Date(2016, 1, 1)),  
    new Article("Body", new Date(2016, 0, 1)),  
    new Article("JavaScript", new Date(2016, 11, 1))  
];  
  
articles.sort(Article.compare);  
  
alert(articles[0].title); // Body
```

Here `Article.compare` stands “over” the 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. Imagine, we need few ways to create an article:

1. Create by given parameters (`title`, `date` etc).
2. Create an empty article with today's date.
3. ...

The first way can be implemented by the constructor. And for the second one we can make a static method of the class.

Like `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:  
Article.remove({id: 12345});
```

Summary

The basic class syntax looks like this:

```
class MyClass {  
    constructor(...) {  
        // ...  
    }  
    method1(...) {}  
    method2(...) {}  
    get something(...) {}  
    set something(...) {}  
    static staticMethod(...) {}
```

```
// ...
}
```

The value of `MyClass` is a function provided as `constructor`. If there's no `constructor`, then an empty function.

In any case, methods listed in the class declaration become members of its `prototype`, with the exception of static methods that are written into the function itself and callable as `MyClass.staticMethod()`. Static methods are used when we need a function bound to a class, but not to any object of that class.

In the next chapter we'll learn more about classes, including inheritance.

✓ Tasks

Rewrite to class

importance: 5

Rewrite the `Clock` class from prototypes to the modern “class” syntax.

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

[Open a sandbox for the task.](#) ↗

[To solution](#)

Class inheritance, super

Classes can extend one another. There's a nice syntax, technically based on the prototypal inheritance.

To inherit from another class, we should specify "extends" and the parent class before the brackets `{ . . . }`.

Here `Rabbit` inherits from `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} stopped. `);
  }
}
```

```

}

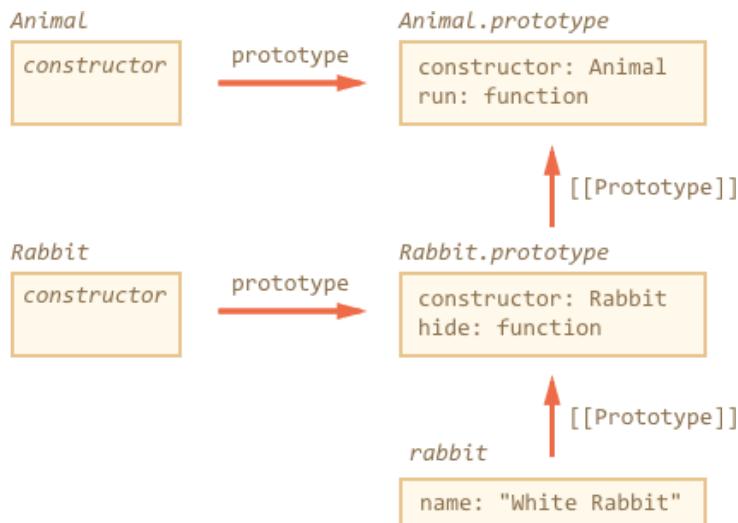
// Inherit from Animal
class Rabbit extends Animal {
  hide() {
    alert(`#${this.name} hides!`);
  }
}

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

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

```

The `extends` keyword actually adds a `[[Prototype]]` reference from `Rabbit.prototype` to `Animal.prototype`, just as you expect it to be, and as we've seen before.



So now `rabbit` has access both to its own methods and to methods of `Animal`.

i 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. As of now, `Rabbit` inherits the `stop` method that sets `this.speed = 0` from `Animal`.

If we specify our own `stop` in `Rabbit`, then it will be used instead:

```
class Rabbit extends Animal {
  stop() {
    // ...this will be used for rabbit.stop()
  }
}
```

...But usually we don't want to totally replace a parent method, but rather to build on top of it, 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} stopped.`);
}

}

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 stopped. White rabbit hides!

```

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

i 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.

Till 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 `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'd really understand what's going on.

In JavaScript, there's a distinction between a "constructor function of an inheriting class" and all others. In an inheriting class, the corresponding constructor function is labelled with a special

```
internal property [[ConstructorKind]] : "derived".
```

The difference is:

- When a normal constructor runs, it creates an empty object as `this` and continues with it.
- But when a derived constructor runs, it doesn't do it. It expects the parent constructor to do this job.

So if we're making a constructor of our own, then we must call `super`, because otherwise the object with `this` reference to it won't be created. And we'll get an error.

For `Rabbit` to work, we need 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
```

Super: internals, [[HomeObject]]

Let's get a little deeper under the hood of `super`. We'll see some interesting things by the way.

First to say, from all that we've learned till now, it's impossible for `super` to work.

Yeah, indeed, let's ask ourselves, how it could technically work? When an object method runs, it gets the current object as `this`. If we call `super.method()` then, how to retrieve the `method`? Naturally, we need to take the `method` from the prototype of the current object. How, technically, we (or a JavaScript engine) can do it?

Maybe we can get the method from `[[Prototype]]` of `this`, as `this.__proto__.method`? Unfortunately, that doesn't work.

Let's try to do it. Without classes, using plain objects for the sake of simplicity.

Here, `rabbit.eat()` should call `animal.eat()` method of the parent object:

```

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:

```
let rabbit = {  
  __proto__: animal,  
  eat() {  
    this.__proto__.eat.call(this); (*)  
  }  rabbit  
};  
  
let longEar = {  
  __proto__: rabbit,  
  eat() {  
    this.__proto__.eat.call(this); (**)  
  }  rabbit  
};
```

1. Inside `longEar.eat()`, the line `(**)` calls `rabbit.eat` providing it with `this=longEar`.

```
// inside longEar.eat() we have this = longEar  
this.__proto__.eat.call(this) // (**)  
// becomes  
longEar.__proto__.eat.call(this)  
// that is  
rabbit.eat.call(this);
```

2. 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);
```

3. ...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.

This actually violates the idea of “unbound” functions, because methods remember their objects. And `[[HomeObject]]` can't be changed, so this bound is forever. So that's a very important change in the language.

But this change is safe. `[[HomeObject]]` is used only for calling parent methods in `super`, to resolve the prototype. So it doesn't break compatibility.

Let's see how it works for `super` – again, using plain objects:

```
let animal = {  
    name: "Animal",  
    eat() { // [[HomeObject]] == animal  
        alert(`${this.name} eats.`);  
    }  
};  
  
let rabbit = {  
    __proto__: animal,  
    name: "Rabbit",  
    eat() { // [[HomeObject]] == rabbit  
        super.eat();  
    }  
};  
  
let longEar = {  
    __proto__: rabbit,  
    name: "Long Ear",  
    eat() { // [[HomeObject]] == longEar  
        super.eat();  
    }  
};  
  
longEar.eat(); // Long Ear eats.
```

Every method remembers its object in the internal `[[HomeObject]]` property. Then `super` uses it to resolve the parent prototype.

`[[HomeObject]]` is defined for methods defined both in classes and in plain objects. But for objects, methods must be specified exactly the given way: as `method()`, not as `"method": function()`.

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() { // should be the short syntax: eat() {...}  
        // ...  
    }  
};  
  
let rabbit = {  
    __proto__: animal,  
    eat: function() {  
        super.eat();  
    }  
};
```

```
};

rabbit.eat(); // Error calling super (because there's no [[HomeObject]])
```

Static methods and inheritance

The `class` syntax supports inheritance for static properties too.

For instance:

```
class Animal {

  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! `);
  }
}

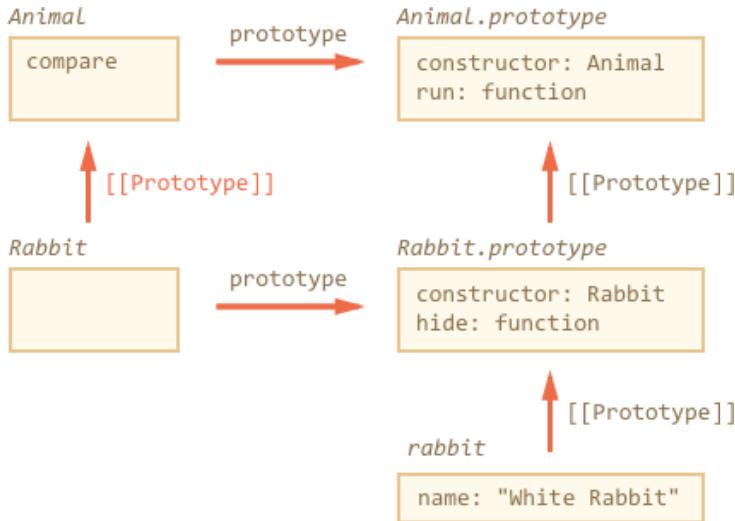
let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Black Rabbit runs with speed 5.
```

Now we can call `Rabbit.compare` assuming that the inherited `Animal.compare` will be called.

How does it work? Again, using prototypes. As you might have already guessed, `extends` also gives `Rabbit` the `[[Prototype]]` reference to `Animal`.



So, `Rabbit` function now inherits from `Animal` function. And `Animal` function normally has `[[Prototype]]` referencing `Function.prototype`, because it doesn't extend anything.

Here, let's check that:

```

class Animal {}
class Rabbit extends Animal {}

// for static properties and methods
alert(Rabbit.__proto__ === Animal); // true

// and the next step is Function.prototype
alert(Animal.__proto__ === Function.prototype); // true

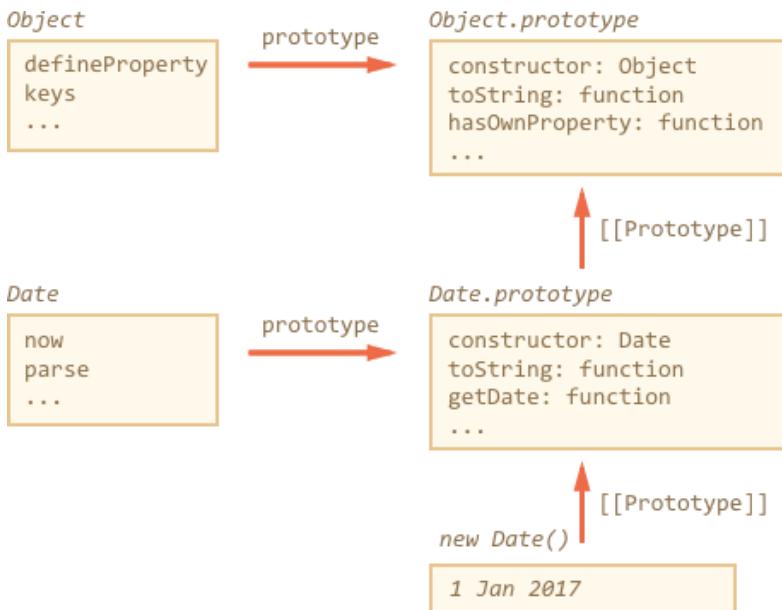
// that's in addition to the "normal" prototype chain for object methods
alert(Rabbit.prototype.__proto__ === Animal.prototype);
  
```

This way `Rabbit` has access to all static methods of `Animal`.

No static inheritance in built-ins

Please note that built-in classes don't have such static `[[Prototype]]` reference. For instance, `Object` has `Object.defineProperty`, `Object.keys` and so on, but `Array`, `Date` etc do not inherit them.

Here's the picture structure for `Date` and `Object`:



Note, there's no link between `Date` and `Object`. Both `Object` and `Date` exist independently. `Date.prototype` inherits from `Object.prototype`, but that's all.

Such difference exists for historical reasons: there was no thought about class syntax and inheriting static methods at the dawn of JavaScript language.

Natives are extendable

Built-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 one very interesting thing. Built-in methods like `filter`, `map` and others – return new objects of exactly the inherited type. They rely on the `constructor` property to do so.

In the example above,

```
arr.constructor === PowerArray
```

So when `arr.filter()` is called, it internally creates the new array of results exactly as `new PowerArray`. And we can keep using its methods further down the chain.

Even more, we can customize that behavior. The static getter `Symbol.species`, if exists, returns the constructor to use in such cases.

For example, here due to `Symbol.species` built-in methods like `map`, `filter` will return “normal” arrays:

```
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
```

We can use it in more advanced keys to strip extended functionality from resulting values if not needed. Or, maybe, to extend it even further.

✓ Tasks

Error creating an instance

importance: 5

Here's the code with `Rabbit` extending `Animal`.

Unfortunately, `Rabbit` objects can't be created. What's wrong? Fix it.

```
class Animal {

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

class Rabbit extends Animal {
  constructor(name) {
    this.name = name;
    this.created = Date.now();
  }
}
```

```
let rabbit = new Rabbit("White Rabbit"); // Error: this is not defined
alert(rabbit.name);
```

[To solution](#)

Extended clock

importance: 5

We've got a `Clock` class. As of now, it prints the time every second.

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.](#) ↗

[To solution](#)

Class extends Object?

importance: 5

As 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
// rabbit.__proto__ === 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') ); // true
```

[To solution](#)

Class 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, here we'll use it for building a *polymorphic* function, the one that treats arguments differently depending on their type.

The `instanceof` operator

The 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` prototypally inherits from `Object`.

The `instanceof` operator examines the prototype chain for the check, and is also fine-tunable using the static method `Symbol.hasInstance`.

The algorithm of `obj instanceof Class` works roughly as follows:

1. If there's a static method `Symbol.hasInstance`, then use it. Like this:

```
// assume anything that canEat 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
```

2. Most classes do not have `Symbol.hasInstance`. In that case, check if `Class.prototype` equals to one of prototypes in the `obj` prototype chain.

In other words, compare:

```
obj.__proto__ === Class.prototype
obj.__proto__.__proto__ === Class.prototype
obj.__proto__.__proto__.__proto__ === Class.prototype
...
```

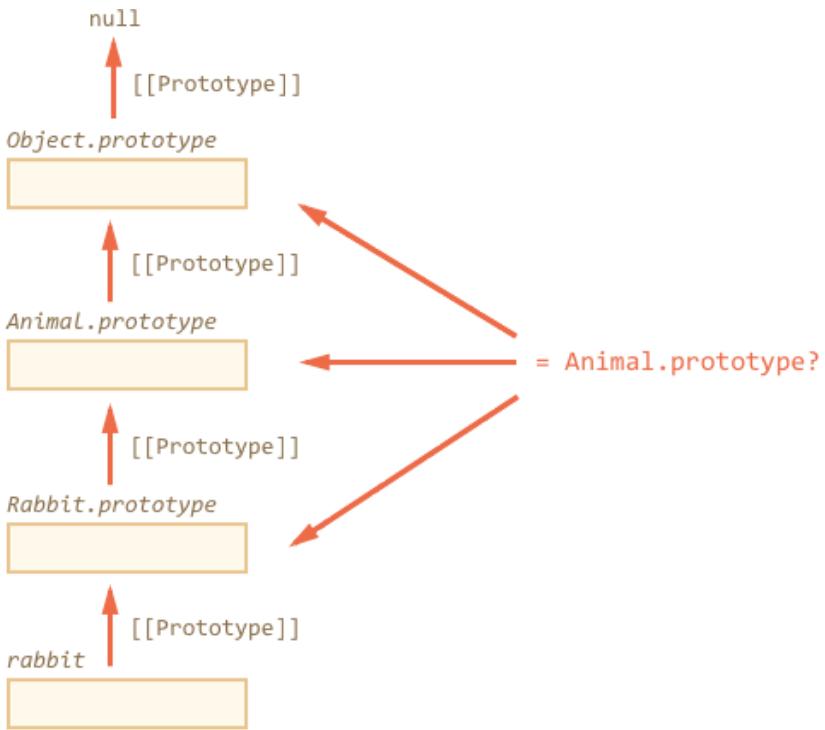
In the example above `Rabbit.prototype === rabbit.__proto__`, so that gives the answer immediately.

In the case of an inheritance, `rabbit` is an instance of the parent class as well:

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // true
// rabbit.__proto__ === Rabbit.prototype
// 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)`.

That'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 `prototype` is changed.

Like here:

```

function Rabbit() {}
let rabbit = new Rabbit();

// changed the prototype
Rabbit.prototype = {};

// ...not a rabbit any more!
alert( rabbit instanceof Rabbit ); // false

```

That's one of the reasons to avoid changing `prototype`. Just to keep safe.

Bonus: Object `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.toStringTag

The 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 few 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.

It can be used instead of `instanceof` for built-in objects when we want to get the type as a string rather than just to check.

Summary

Let's recap the type-checking methods that we know:

	works for	returns
<code>typeof</code>	primitives	string
<code>{ } .toString</code>	primitives, built-in objects, objects with <code>Symbol.toStringTag</code>	string
<code>instanceof</code>	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.

Tasks

Strange `instanceof`

importance: 5

Why `instanceof` below returns `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
```

[To solution](#)

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, I have a class `StreetSweeper` and a class `Bicycle`, and want to make a `StreetSweepingBicycle`.

Or, talking about programming, we have a class `Renderer` that implements templating and a class `EventEmitter` that implements event handling, and want to merge these functionalities together with a class `Page`, to make a page that can use templates and emit events.

There's a concept that can help here, called "mixins".

As defined in Wikipedia, a [mixin ↗](#) is a class that contains methods for use by other classes without having to be the parent class of those other classes.

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 make 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 extend some other class and also include the mixin to "mix-in" the additional methods, like this:

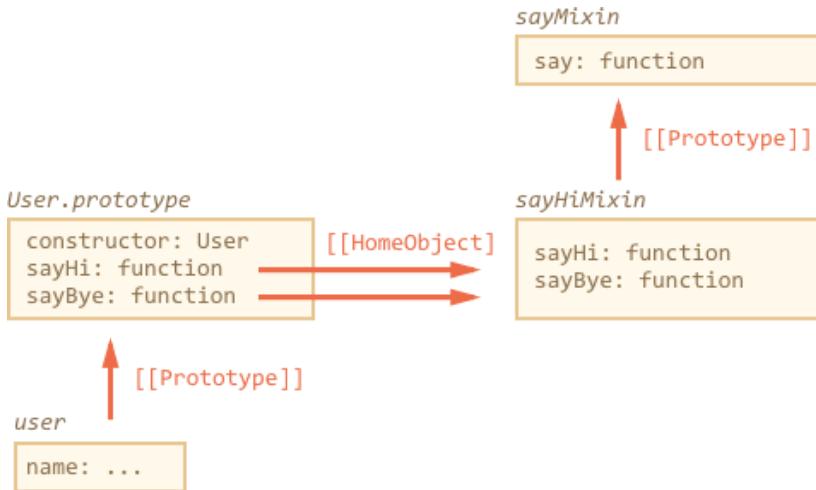
```
class User extends Person {  
    // ...  
}  
  
Object.assign(User.prototype, sayHiMixin);
```

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.create 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` looks for the method in the prototype of that mixin, not the class.



That's because methods from `sayHiMixin` have `[[HomeObject]]` set to it. So `super` actually means `sayHiMixin.__proto__`, not `User.__proto__`.

EventMixin

Now let's make a mixin for real life.

The important feature of many objects is working with events.

That is: an object should have a method to "generate an event" when something important happens to it, and other objects should be able to "listen" to such events.

An event must have a name and, optionally, bundle some additional data.

For instance, an object `user` can generate an event `"login"` when the visitor logs in. And another object `calendar` may want to receive 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 want to get that information and react on that event.

Events is a way to "share information" with anyone who wants it. They can be useful in any class, so let's make a mixin for them:

```
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 && this._eventHandlers[eventName];
if (!handlers) return;
for (let i = 0; i < handlers.length; i++) {
  if (handlers[i] === handler) {
    handlers.splice(i--, 1);
  }
}
},
/** 
 * Generate the event and attach the data to it
 * this.trigger('select', data1, data2);
 */
trigger(eventName, ...args) {
  if (!this._eventHandlers || !this._eventHandlers[eventName]) {
    return; // no handlers for that event name
  }

  // call the handlers
  this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
}
};

```

There are 3 methods here:

1. `.on(eventName, handler)` – assigns function `handler` to run when the event with that name happens. The handlers are stored in the `_eventHandlers` property.
2. `.off(eventName, handler)` – removes the function from the handlers list.
3. `.trigger(eventName, ...args)` – generates the event: all assigned handlers are called and `args` are passed as arguments to them.

Usage:

```

// Make a class
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Add the mixin
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// call the handler on selection:
menu.on("select", value => alert(`Value selected: ${value}`));

// triggers the event => shows Value selected: 123
menu.choose("123"); // value selected

```

Now if we have the code interested to react on user selection, we can bind it with `menu.on(...)`.

And the `eventMixin` can add such behavior to as many classes as we'd like, without interfering with the inheritance chain.

Summary

Mixin – is a generic object-oriented programming term: a class that contains methods for other classes.

Some other languages like e.g. python allow to create mixins using multiple inheritance. JavaScript does not support multiple inheritance, but mixins can be implemented by copying them into the prototype.

We can use mixins as a way to augment a class by multiple behaviors, like event-handling as we have seen above.

Mixins may become a point of conflict if they occasionally overwrite native class methods. So generally one should think well about the naming for a mixin, to minimize such possibility.

Error handling

Error 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 of 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 to “catch” errors and, instead of dying, do something more reasonable.

The “try...catch” syntax

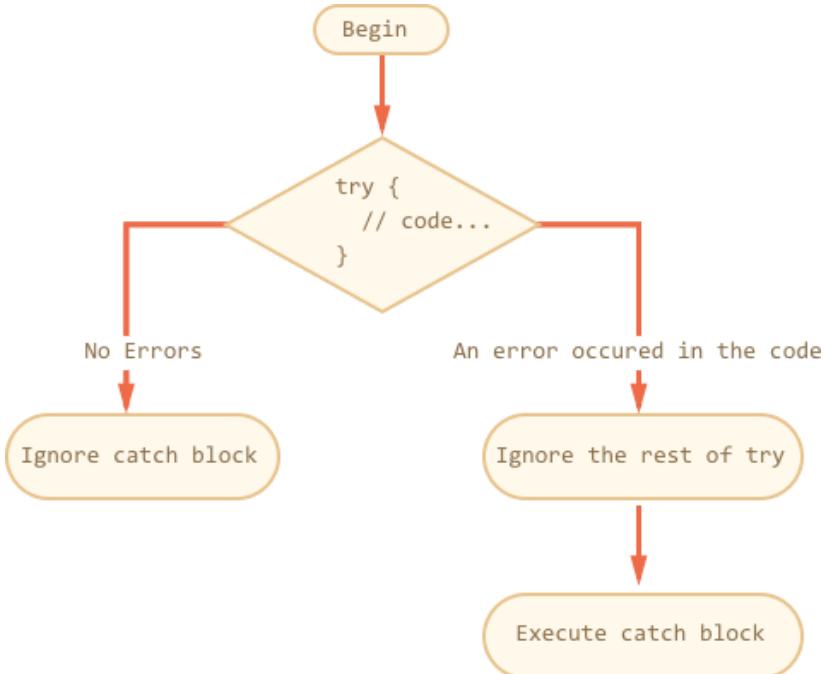
The `try..catch` construct has two main blocks: `try`, and then `catch`:

```
try {  
  // code...  
}  
  catch (err) {  
    // error handling  
}
```

It works like this:

1. First, the code in `try { . . . }` is executed.
2. If there were no errors, then `catch(err)` is ignored: the execution reaches the end of `try` and then jumps over `catch`.
3. If an error occurs, then `try` execution is stopped, and the control flows to the beginning of `catch(err)`. The `err` variable (can use any name for it) contains an error object with

details about what's happened.



So, an error inside the `try {...}` block does not kill the script: we have a chance to handle it in `catch`.

Let's see more 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)  
}  
alert("...Then the execution continues");
```

- 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) <--  
}  
  
alert("...Then the execution continues");
```

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(e) {  
    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 phrase 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 the 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 (e) {
  alert( "won't work" );
}
```

That’s because `try..catch` actually wraps the `setTimeout` call that schedules the function. But 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 (e) {
    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 inside `catch` block has two main properties:

name

Error name. 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 ...
}

// Can also show an error as a whole
// The error is converted to string as "name: message"
alert(err); // ReferenceError: lalala is not defined
}
```

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 (e) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors, we'll try to request it one more time." );
  alert( e.name );
  alert( e.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 errors

What 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 (e) {
  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” operator

The `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_0");

alert(error.name); // Error
alert(error.message); // Things happen o_0
```

Let's see what kind of error `JSON.parse` generates:

```
try {
  JSON.parse("{ bad json o_0 }");
} catch(e) {
  alert(e.name); // SyntaxError
  alert(e.message); // Unexpected token o in JSON at position 0
}
```

As we can see, that's a `SyntaxError`.

And in our case, the absence of `name` could be treated as a syntax error also, assuming that 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(e) {
  alert( "JSON Error: " + e.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 variable is undefined or something else, not just that “incorrect data” thing.

Like this:

```
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 crazy bug may be discovered that leads to terrible hacks (like it happened with the `ssh` tool).

In our case, `try..catch` is meant 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.

Fortunately, we can find out which error we get, for instance from its `name`:

```
try {
  user = { /*...*/ };
} catch(e) {
  alert(e.name); // "ReferenceError" for accessing an undefined variable
}
```

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:

1. Catch gets all errors.
2. In `catch(err) { ... }` block we analyze the error object `err`.
3. If we don't know how to handle it, then we do `throw err`.

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(e) {

  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // 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 (e) {
    // ...
    if (e.name != 'SyntaxError') {
      throw e; // rethrow (don't know how to deal with it)
    }
  }
}

try {
  readData();
} catch (e) {
  alert( "External catch got: " + e ); // caught it!
}
```

Here `readData` only knows how to handle `SyntaxError`, while the outer `try..catch` knows how to handle everything.

try...catch...finally

Wait, 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(e) {  
    ... handle errors ...  
} finally {  
    ... execute always ...  
}
```

Try running this code:

```
try {  
    alert( 'try' );  
    if (confirm('Make an error?')) BAD_CODE();  
} catch (e) {  
    alert( 'catch' );  
} finally {  
    alert( 'finally' );  
}
```

The code has two ways of execution:

1. If you answer “Yes” to “Make an error?”, then `try -> catch -> finally`.
2. If you say “No”, then `try -> finally`.

The `finally` clause is often used when we start doing something before `try..catch` 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 (e) {
  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, there may be two ways to exit a function: either a `return` or `throw`. The `finally` clause handles them both.

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 `let` were made inside the `{ . . . }` block, it would only be visible inside of it.

i 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 (e) {  
        /* ... */  
    } finally {  
        alert('finally');  
    }  
  
    alert( func() ); // first works alert from finally, and then this one  
}
```

i try..finally

The `try..finally` construct, without `catch` clause, is also useful. We apply it when we don't want to handle errors right here, 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 jumps outside.

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 something else terrible.

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 special `window.onerror` ↗ property. It 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 <http://www.muscula.com> ↗ .

They work like this:

1. We register at the service and get a piece of JS (or a script URL) from them to insert on pages.
2. That JS script has a custom `window.onerror` function.
3. When an error occurs, it sends a network request about it to the service.
4. 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 `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) – the stack at the moment of error creation.

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 basic 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 to setup a "global" error handler to catch errors that "fall out". In-browser that's `window.onerror`.

Tasks

Finally or just the code?

importance: 5

Compare the two code fragments.

1. The first one uses `finally` to execute the code after `try..catch`:

```
try {
  work work
} catch (e) {
  handle errors
} finally {
  cleanup the working space
}
```

2. The second fragment puts the cleaning right after `try..catch`:

```
try {
  work work
} catch (e) {
  handle errors
}

cleanup the working space
```

We definitely need the cleanup after the work has started, 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.

[To solution](#)

Custom errors, extending Error

When 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 `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 we build our application, our own errors naturally form a hierarchy, for instance `HttpTimeoutError` may inherit from `HttpError`, and so on.

Extending Error

As 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 built-in `Error` class.

That class is built-in, but we should have its approximate code before our eyes, to understand what we're extending.

So here you are:

```
// 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 = <nested calls>; // non-standard, but most environments support it
  }
}
```

Now let's go on and inherit `ValidationError` from it:

```
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 take a look at the constructor:

1. 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.
2. 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` only knows how to handle validation and syntax errors, other kinds (due to a typo in the code or such) 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`). 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>` when creating each custom error. But there's a way out. We can make our own “basic error” class that removes this burden from our shoulders by using `this.constructor.name` for `this.name` in the constructor. And then inherit 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”, right? 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: the new code will probably generate other kinds of errors.

The code which calls `readUser` should handle these errors. Right now it uses multiple `if` in the `catch` block to check for different error types and rethrow the unknown ones. But if

`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 in every code that calls `readUser`?

Often the answer is “No”: the outer code wants to be “one level above all that”. It wants to have some kind of “data reading error”. Why exactly it happened – is often irrelevant (the error message describes it). Or, even better if there is a way to get error details, but only if we need to.

So let's make a new class `ReadError` to represent such errors. If an error occurs inside `readUser`, we'll catch it there and generate `ReadError`. We'll also keep the reference to the original error in its `cause` property. Then the outer code will only have to check for `ReadError`.

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 possible all error types.

The approach is called “wrapping exceptions”, because we take “low level exceptions” and “wrap” them into `ReadError` that is more abstract and more convenient to use for the calling code. It is widely used in object-oriented programming.

Summary

- We can inherit from `Error` and other built-in error classes normally, just need to take care of `name` property and don't forget to call `super`.
- Most of the time, we should use `instanceof` to check for particular errors. It also works with inheritance. But sometimes we have an error object coming from the 3rd-party library and there's no easy way to get the class. Then `name` property can be used for such checks.
- Wrapping exceptions is a widespread technique when a function handles low-level exceptions and makes a higher-level object to report about the errors. Low-level exceptions sometimes become properties of that object like `err.cause` in the examples above, but that's not strictly required.

Tasks

Inherit from `SyntaxError`

importance: 5

Create 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)
```

[To solution](#)

Solutions

Hello, world!

Show an alert

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Show an alert with an external script

The 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!");
```

[To formulation](#)

Variables

Working with variables

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"
```

To formulation

Giving the right name

First, the variable for the name of our planet.

That's simple:

```
let ourPlanetName = "Earth";
```

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

Second, the name of the current visitor:

```
let currentUserName = "John";
```

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

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

And if your editor does not have proper autocompletion, get a [new one](#).

To formulation

Uppercase const?

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.

To formulation

Data types

String quotes

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

```
let name = "Ilya";

// the expression is a number 1
alert(`hello ${1}`); // hello 1

// the expression is a string "name"
alert(`hello ${"name"} `); // hello name

// the expression is a variable, embed it
alert(`hello ${name}`); // hello Ilya
```

To formulation

Type Conversions

Type conversions

```
"" + 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
7 / 0 = Infinity
"-9\n" + 5 = " -9\n5"
"-9\n" - 5 = -14
null + 1 = 1 // (3)
undefined + 1 = NaN // (4)
```

1. The addition with a string `"" + 1` converts `1` to a string: `"" + 1 = "1"`, and then we have `"1" + 0`, the same rule is applied.
2. The subtraction `-` (like most math operations) only works with numbers, it converts an empty string `""` to `0`.
3. `null` becomes `0` after the numeric conversion.
4. `undefined` becomes `NaN` after the numeric conversion.

To formulation

Operators

The postfix and prefix forms

The 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
```

To formulation

Assignment result

The answer is:

- a = 4 (multiplied by 2)
- x = 5 (calculated as 1 + 4)

To formulation

Comparisons

Comparisons

```
5 > 4 → true
"apple" > "pineapple" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

Some of the reasons:

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

[To formulation](#)

Interaction: alert, prompt, confirm

A simple page

JavaScript-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>
```

[To formulation](#)

Conditional operators: if, '?'

if (a string with zero)

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' );
}
```

To formulation

The name of JavaScript

```
<!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>
```

To formulation

Show the sign

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

if (value > 0) {
    alert( 1 );
} else if (value < 0) {
    alert( -1 );
} else {
    alert( 0 );
}
```

To formulation

Check the login

```
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.

To formulation

Rewrite 'if' into '?

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

To formulation

Rewrite 'if..else' into '?

```
let message = (login == 'Employee') ? 'Hello' :
    (login == 'Director') ? 'Greetings' :
    (login == '') ? 'No login' :
    '';
```

To formulation

Logical operators

What's the result of OR?

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

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

[To formulation](#)

What's the result of OR'ed alerts?

The answer: first `1`, then `2`.

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

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

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

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

[To formulation](#)

What is the result of AND?

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

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

[To formulation](#)

What is the result of AND'ed alerts?

The answer: `1`, and then `undefined`.

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

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

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

it's done.

To formulation

The result of OR AND OR

The answer: 3 .

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

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

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

```
null || 3 || 4
```

Now the result is the first truthy value: 3 .

To formulation

Check the range between

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

To formulation

Check the range outside

The first variant:

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

The second variant:

```
if (age < 14 || age > 90)
```

To formulation

A question about "if"

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' );
```

To formulation

Loops: while and for

Last loop value

The answer: 1.

```
let i = 3;  
  
while (i) {  
    alert( i-- );  
}
```

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

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

```
let i = 3;  
  
alert(i--); // shows 3, decreases i to 2  
  
alert(i--); // shows 2, decreases i to 1  
  
alert(i--); // shows 1, decreases i to 0  
  
// done, while(i) check stops the loop
```

To formulation

Which values does the while loop show?

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

1. From 1 to 4

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

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

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

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

2. From 1 to 5

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

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

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

Then follow `2, 3, 4...`

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

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

[To formulation](#)

Which values get shown by the "for" loop?

The answer: from 0 to 4 in both cases.

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

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

That can be easily deducted from the algorithm of `for`:

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

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

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

To formulation

Output even numbers in the loop

```
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.

To formulation

Replace "for" with "while"

```
let i = 0;  
while (i < 3) {  
    alert(`number ${i}!`);  
    i++;  
}
```

To formulation

Repeat until the input is correct

```
let num;  
  
do {  
    num = prompt("Enter a number greater than 100?", 0);  
} while (num <= 100 && num);
```

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

1. The check for `num <= 100` – that is, the entered value is still not greater than `100`.

2. The check `&& num` is false when `num` is `null` or an empty string. Then the `while` loop stops too.

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

[To formulation](#)

Output prime numbers

There are many algorithms for this task.

Let's use a nested loop:

```
For each i in the interval {
    check if i has a divisor from 1..i
    if yes => the value is not a prime
    if no => the value is a prime, show it
}
```

The code using a label:

```
let n = 10;

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

    for (let j = 2; j < i; j++) { // look for a divisor..
        if (i % j == 0) continue nextPrime; // not a prime, go next i
    }

    alert( i ); // a prime
}
```

There's a lot of space to optimize it. For instance, we could look for the divisors from 2 to square root of `i`. But anyway, if we want to be really efficient for large intervals, we need change the approach and rely on advanced maths and complex algorithms like [Quadratic sieve ↗](#), [General number field sieve ↗](#) etc.

[To formulation](#)

The "switch" statement

Rewrite the "switch" into an "if"

To 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.

To formulation

Rewrite "if" into "switch"

The 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.

To formulation

Functions

Is "else" required?

No difference.

To formulation

Rewrite the function using '?' or '||'

Using a question mark operator '?':

```
function checkAge(age) {
    return (age > 18) ? true : confirm('Did parents allow you?');
}
```

Using OR || (the shortest variant):

```
function checkAge(age) {
    return (age > 18) || confirm('Did parents allow you?');
}
```

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

To formulation

Function min(a, b)

A solution using `if`:

```
function min(a, b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

A solution with a question mark operator '?':

```
function min(a, b) {
    return a < b ? a : b;
}
```

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

To formulation

Function pow(x,n)

```
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 an integer greater than 0`);
} else {
  alert( pow(x, n) );
}
```

To formulation

Function expressions and arrows

Rewrite with arrow functions

```
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?

To formulation

Coding Style

Bad style

You could note the following:

```
function pow(x,n) // <- no space between arguments
{ // <- figure bracket on a separate line
  let result=1; // <- no spaces to the both sides of =
  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 ;
if (n<0) // <- no spaces inside (n < 0), and should be extra line above it
{ // <- figure bracket on a separate line
  // below - a long line, may be worth to split into 2 lines
  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 ;
}
```

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) );
}
```

To formulation

Automated testing with mocha

What's wrong in the test?

The 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 inside 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);
  });
});
```

[To formulation](#)

Objects

Hello, object

```
let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

To formulation

Check for emptiness

Just loop over the object and `return false` immediately if there's at least one property.

```
function isEmpty(obj) {
  for (let key in obj) {
    return false;
  }
  return true;
}
```

Open the solution with tests in a sandbox. ↗

To formulation

Constant objects?

Sure, it works, no problem.

The `const` only protects the variable itself from changing.

In other words, `user` stores a reference to the object. And it can't be changed. But the content of the object can.

```
const user = {
  name: "John"
};

// works
user.name = "Pete";

// error
user = 123;
```

To formulation

Sum object properties

```
let salaries = {  
    John: 100,  
    Ann: 160,  
    Pete: 130  
};  
  
let sum = 0;  
for (let key in salaries) {  
    sum += salaries[key];  
}  
  
alert(sum); // 390
```

To formulation

Multiply numeric properties by 2

Open the solution with tests in a sandbox. ↗

To formulation

Object methods, "this"

Syntax check

Error!

Try it:

```
let user = {  
    name: "John",  
    go: function() { alert(this.name) }  
}  
  
(user.go)() // error!
```

The error message in most browsers does not give understanding what went wrong.

The error appears because a semicolon is missing after `user = {...}`.

JavaScript does not assume 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 brackets 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.

[To formulation](#)

Explain the value of "this"

Here's the explanations.

1. That's a regular object method call.
2. The same, brackets do not change the order of operations here, the dot is first anyway.
3. Here we have a more complex call `(expression).method()`. 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`.

4. The similar thing as (3), to the left of the dot `.` 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`.

[To formulation](#)

Using "this" in object literal

Answer: 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 literals.

Here the value of `this` inside `makeUser()` is `undefined`, because it is called as a function, not as a method.

And the object literal itself has no effect on `this`. 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.

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 `.`.

[To formulation](#)

Create a calculator

```
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.](#)

[To formulation](#)

Chaining

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().up().down().showStep(); // 1

```

We also can write a single call per line. For long chains it's more readable:

```

ladder
    .up()
    .up()
    .down()
    .up()
    .down()
    .showStep(); // 1

```

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Constructor, operator "new"

Two functions – one object

Yes, 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
```

[To formulation](#)

Create new Calculator

```
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.](#) ↗

[To formulation](#)

Create new Accumulator

```
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.](#) ↗

[To formulation](#)

Methods of primitives

Can I add a string property?

Try running it:

```
let str = "Hello";

str.test = 5; // (*) 

alert(str.test);
```

There may be two kinds of result:

1. `undefined`
2. An error.

Why? Let's replay what's happening at line `(*)`:

1. When a property of `str` is accessed, a “wrapper object” is created.
2. The operation with the property is carried out on it. So, the object gets the `test` property.
3. The operation finishes and the “wrapper object” disappears.

So, on the last line, `str` has no trace of the property. A new wrapper object for every object operation on a string.

Some browsers though may decide to further limit the programmer and disallow to assign properties to primitives at all. That's why in practice we can also see errors at line `(*)`. It's a little bit farther from the specification though.

This example clearly shows that primitives are not objects.

They just can not store data.

All property/method operations are performed with the help of temporary objects.

[To formulation](#)

Numbers

Sum numbers from the visitor

```
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"`.

[To formulation](#)

Why `6.35.toFixed(1) == 6.3?`

Internally 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) ); // 63.5000000000000000000000
```

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
```

[To formulation](#)

Repeat until the input is a number

```
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.](#) ↗

[To formulation](#)

An occasional infinite loop

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.

To formulation

A random number from min to max

We need to “map” all values from the interval 0...1 into values from min to max.

That can be done in two stages:

1. If we multiply a random number from 0...1 by max-min , then the interval of possible values increases 0..1 to 0..max-min .
2. 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) );
```

To formulation

A random integer from min to max

The simple but wrong solution

The 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

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.

[To formulation](#)

Strings

Uppercast the first character

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, so we'll get an error.

There are two variants here:

1. Use `str.charAt(0)`, as it always returns a string (maybe empty).
2. Add a test for an empty string.

Here's the 2nd variant:

```
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.](#) ↗

[To formulation](#)

Check for spam

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 xxxx') );  
alert( checkSpam("innocent rabbit") );
```

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Truncate the text

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.](#)

[To formulation](#)

Extract the money

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Arrays

Is array copied?

The 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.

[To formulation](#)

Array operations.

```
let styles = ["Jazz", "Blues"];
styles.push("Rock-n-Roll");
styles[Math.floor((styles.length - 1) / 2)] = "Classics";
alert( styles.shift() );
styles.unshift("Rap", "Reggie");
```

To formulation

Calling in an array context

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.

To formulation

Sum input numbers

Please 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() );

```

To formulation

A maximal subarray

The slow solution

We 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 a 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 detail 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.](#)

[To formulation](#)

Array methods

Translate border-left-width to borderLeftWidth

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Filter range

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Filter range "in place"

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Sort in the reverse order

```
let arr = [5, 2, 1, -10, 8];  
arr.sort((a, b) => b - a);  
alert( arr );
```

[To formulation](#)

Copy and sort array

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 );
```

To formulation

Create an extendable calculator

- Please note how methods are stored. They are simply added to the internal object.
- All tests and numeric conversions are done in the `calculate` method. In future it may be extended to support more complex expressions.

[Open the solution with tests in a sandbox.](#) ↗

To formulation

Map to names

```
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
```

To formulation

Map to objects

```
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 for 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.

[To formulation](#)

Sort objects

```
function sortByName(arr) {
  arr.sort((a, b) => a.name > b.name);
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ john, pete, mary ];

sortByName(arr);

// now sorted is: [john, mary, pete]
alert(arr[1].name); // Mary
```

[To formulation](#)

Shuffle an array

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 (for V8, July 2017):

```

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
    [array[i], array[j]] = [array[j], array[i]]; // swap elements
  }
}
```

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 < 10000000; 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.

[To formulation](#)

Get average age

```
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
```

[To formulation](#)

Filter unique array members

Let'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", ":-0"  
];  
  
alert( unique(strings) ); // Hare, Krishna, :-0
```

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 $10000 * 10000 = 100$ millions of comparisons. That's a lot.

So the solution is only good for small arrays.

Further in the chapter [Map, Set, WeakMap and WeakSet](#) we'll see how to optimize it.

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Map, Set, WeakMap and WeakSet

Filter unique array members

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Filter anagrams

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 = arr[i] // 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 Array.from(Object.values(obj));
}

```

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];  
alert( aclean(arr) );
```

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Iterable keys

That'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
```

[To formulation](#)

Store "unread" flags

The sane choice here is a `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. We can't get "all read messages" directly. But we can do it by iterating over all messages and filtering those that are in the set.

P.S. Adding a property of our own to each message may be dangerous if messages are managed by someone else's code, but we can make it a symbol to evade conflicts.

Like this:

```
// the symbolic property is only known to our code
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Now even if someone else's code uses `for .. in` loop for message properties, our secret flag won't appear.

[To formulation](#)

Store read dates

To 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
```

[To formulation](#)

Object.keys, values, entries

Sum the properties

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Count properties

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Destructuring assignment

Destructuring assignment

```
let user = {  
    name: "John",  
    years: 30  
};  
  
let {name, years: age, isAdmin = false} = user;  
  
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

[To formulation](#)

The maximal salary

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Date and time

Create a date

The `new Date` constructor uses the local time zone by default. So the only important thing to remember is that months start from zero.

So February has number 1.

```
let d = new Date(2012, 1, 20, 3, 12);  
alert( d );
```

[To formulation](#)

Show a weekday

The 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.](#) ↗

[To formulation](#)

European weekday

```
function getLocalDay(date) {

  let day = date.getDay();

  if (day == 0) { // 0 becomes 7
    day = 7;
  }

  return day;
}

alert( getLocalDay(new Date(2012, 0, 3)) ); // 2
```

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Which day of month was many days ago?

The 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.](#)

[To formulation](#)

Last day of month?

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.](#)

[To formulation](#)

How many seconds has passed today?

To 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();
}

```

To formulation

How many seconds till tomorrow?

To 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.

To formulation

Format the relative date

To 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;

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.](#) ↗

[To formulation](#)

JSON methods, toJSON

Turn the object into JSON and back

```

let user = {
  name: "John Smith",
  age: 35
};

let user2 = JSON.parse(JSON.stringify(user));

```

[To formulation](#)

Exclude backreferences

```

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`.

[To formulation](#)

Recursion and stack

Sum all numbers till the given one

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: `sumTo(n) = n*(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. The standard describes a “tail call” optimization: if the recursive call is the very last one in the function (like in `sumTo` above), then the outer function will not need to resume the execution and we don't need to remember its execution context. In that case `sumTo(100000)` is countable. But if your JavaScript engine does not support it, there will be an error: maximum stack size exceeded, because there's usually a limitation on the total stack size.

[To formulation](#)

Calculate factorial

By definition, a factorial is `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
```

[To formulation](#)

Fibonacci numbers

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, `fib(77)` may hang up the engine for some time eating all CPU resources.

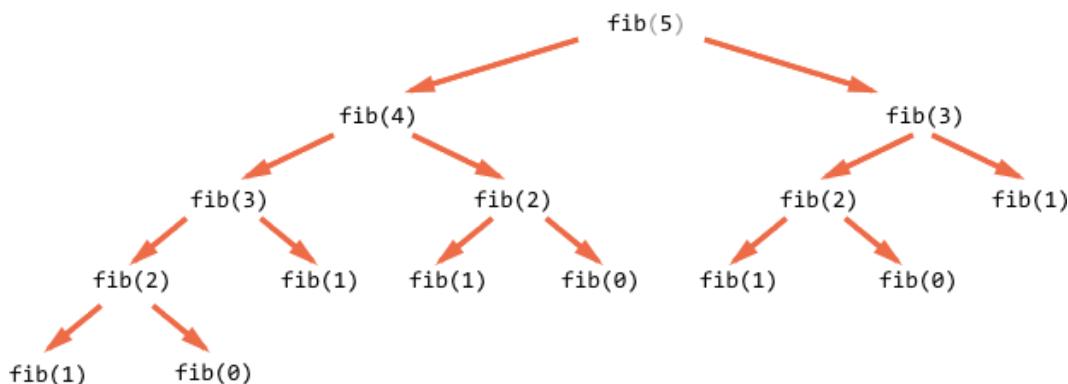
That's because the function makes too many subcalls. The same values are re-evaluated again and again.

For instance, let's see a piece of calculations for `fib(5)`:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

Here we can see that the value of `fib(3)` is needed for both `fib(5)` and `fib(4)`. So `fib(3)` will be called and evaluated two times completely independently.

Here's the full recursion tree:



We can clearly notice that `fib(3)` is evaluated two times and `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 `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 `fib(3)` as their sum, then `fib(4)` as the sum of two previous values, then `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 `fib(4) = fib(2) + fib(3)`.

Let's shift the variables: `a, b` will get `fib(2), 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 ↗](#).

To formulation

Output a single-linked list

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

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.

To formulation

Output a single-linked list in the reverse order

Using a recursion

The 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 loop

The 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.

[To formulation](#)

Closure

Are counters independent?

The 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`.

[To formulation](#)

Counter object

Surely 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

```

To formulation

Function in if

The result is **an error**.

The function `sayHi` is declared inside the `if`, so it only lives inside it. There is no `sayHi` outside.

To formulation

Sum with closures

For the second brackets 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

```

To formulation

Filter through function

Filter 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 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.](#) ↗

[To formulation](#)

Sort by field

```
let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];

function byField(field) {
  return (a, b) => a[field] > b[field] ? 1 : -1;
}

users.sort(byField('name'));
users.forEach(user => alert(user.name)); // Ann, John, Pete

users.sort(byField('age'));
users.forEach(user => alert(user.name)); // Pete, Ann, John
```

[To formulation](#)

Army of functions

Let's examine what's done inside `makeArmy`, and the solution will become obvious.

1. It creates an empty array `shooters`:

```
let shooters = [];
```

2. Fills it in the loop via `shooters.push(function...)`.

Every element is a function, so the resulting array looks like this:

```
shooters = [
  function () { alert(i); },
  function () { alert(i); }
];
```

3. The array is returned from the function.

Then, later, the call to `army[5]()` will get the element `army[5]` from the array (it will be a function) and call it.

Now why all such functions show the same?

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.

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
    };
    ...
  }
  ...
}
```

...We can see that it lives in the lexical environment associated with the current `makeArmy()` run. But when `army[5]()` is called, `makeArmy` has already finished its job, and `i` has the last value: `10` (the end of `while`).

As a result, all `shooter` functions get from the outer lexical environment the same, last value `i=10`.

The fix can be very simple:

```
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
```

Now it works correctly, because every time the code block in `for (. . .) { . . . }` is executed, a new Lexical Environment is created for it, with the corresponding value of `i`.

So, the value of `i` now lives a little bit closer. Not in `makeArmy()` Lexical Environment, but in the Lexical Environment that corresponds the current loop iteration. A `shooter` gets the value exactly from the one where it was created.

```
shooters = [  
    function () { alert(i); }, → i: 0  
    function () { alert(i); }, → i: 1 → makeArmy()  
    function () { alert(i); }, → i: 2 → outer Lexical Environment  
    ...  
    function () { alert(i); } → i: 10  
];
```

Here we rewrote `while` into `for`.

Another trick could be possible, let's see it for better understanding of the subject:

```
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();

army[0](); // 0
army[5](); // 5
```

The `while` loop, just like `for`, makes a new Lexical Environment for each run. So here we make sure that it gets the right value for a `shooter`.

We copy `let j = i`. This makes a loop body local `j` and copies the value of `i` to it. Primitives are copied “by value”, so we actually get a complete independent copy of `i`, belonging to the current loop iteration.

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Function object, NFE

Set and decrease for counter

The 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`.

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Sum with an arbitrary amount of brackets

1. For the whole thing to work *anyhow*, the result of `sum` must be function.
2. That function must keep in memory the current value between calls.
3. 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, so 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.

To formulation

Scheduling: `setTimeout` and `setInterval`

Output every second

Using `setInterval`:

```

function printNumbers(from, to) {
    let current = from;

```

```

let timerId = setInterval(function() {
  alert(current);
  if (current == to) {
    clearInterval(timerId);
  }
  current++;
}, 1000);
}

// usage:
printNumbers(5, 10);

```

Using recursive `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. Sometimes we need to add a line to make the first output immediately, that's easy to do.

To formulation

Rewrite `setTimeout` with `setInterval`

```

let i = 0;

let start = Date.now();

let timer = setInterval(count, 0);

function count() {

  for(let j = 0; j < 10000000; j++) {
    i++;
  }

  if (i == 100000000) {
    alert("Done in " + (Date.now() - start) + 'ms');
    clearInterval(timer);
  }
}

```

To formulation

What will setTimeout show?

Any `setTimeout` will run only after the current code has finished.

The `i` will be the last one: `1000000000`.

```
let i = 0;

setTimeout(() => alert(i), 100); // 1000000000

// assume that the time to execute this function is >100ms
for(let j = 0; j < 1000000000; j++) {
    i++;
}
```

To formulation

Decorators and forwarding, call/apply

Spy decorator

Here we can use `calls.push(args)` to store all arguments in the log and `f.apply(this, args)` to forward the call.

[Open the solution with tests in a sandbox.](#)

To formulation

Delaying decorator

The solution:

```
function delay(f, ms) {

    return function() {
        setTimeout(() => f.apply(this, arguments), ms);
    };
}
```

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` (in-browser), so we'd need to write a bit more code to pass them from the wrapper:

```
function delay(f, ms) {  
  
    // added variables to pass this and arguments from the wrapper inside setTimeout  
    return function(...args) {  
        let savedThis = this;  
        setTimeout(function() {  
            f.apply(savedThis, args);  
        }, ms);  
    };  
}
```

[Open the solution with tests in a sandbox.](#)

[To formulation](#)

Debounce decorator

```
function debounce(f, ms) {  
  
    let isCooldown = false;  
  
    return function() {  
        if (isCooldown) return;  
  
        f.apply(this, arguments);  
  
        isCooldown = true;  
  
        setTimeout(() => isCooldown = false, ms);  
    };  
}
```

The call to `debounce` returns a wrapper. There may be two states:

- `isCooldown = false` – ready to run.
- `isCooldown = true` – waiting for the timeout.

In the first call `isCooldown` is falsy, so the call proceeds, and the state changes to `true`.

While `isCooldown` is true, all other calls are ignored.

Then `setTimeout` reverts it to `false` after the given delay.

[Open the solution with tests in a sandbox.](#)

Throttle decorator

```
function throttle(func, ms) {  
  
  let isThrottled = false,  
      savedArgs,  
      savedThis;  
  
  function wrapper() {  
  
    if (isThrottled) { // (2)  
      savedArgs = arguments;  
      savedThis = this;  
      return;  
    }  
  
    func.apply(this, arguments); // (1)  
  
    isThrottled = true;  
  
    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`.

1. During the first call, the `wrapper` just runs `func` and sets the cooldown state (`isThrottled = true`).
2. In this state all calls 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.
3. ...Then after `ms` milliseconds pass, `setTimeout` triggers. The cooldown state is removed (`isThrottled = false`). And if we had ignored calls, then `wrapper` is executed with 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.](#) ↗

Function binding

Bound function as a method

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`.

To formulation

Second bind

The 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.

To formulation

Function property after bind

The answer: `undefined`.

The result of `bind` is another object. It does not have the `test` property.

To formulation

Ask losing this

The 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, but may fail in more complex situations where `user` has a chance of being overwritten between the moments of asking and running `() => user.loginOk()`.

To formulation

Currying and partials

Partial application for login

1. 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.

2. 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));
```

[To formulation](#)

Prototypal inheritance

Working with prototype

1. `true`, taken from `rabbit`.
2. `null`, taken from `animal`.
3. `undefined`, there's no such property any more.

[To formulation](#)

Searching algorithm

1. 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
```

2. 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.

To formulation

Where it writes?

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`

To formulation

Why two hamsters are full?

Let's look carefully at what's going on in the call `speedy.eat("apple")`.

1. The method `speedy.eat` is found in the prototype (=hamster), then executed with `this=speedy` (the object before the dot).
2. Then `this.stomach.push()` needs to find `stomach` property and call `push` on it. It looks for `stomach` in `this` (=speedy), but nothing found.
3. Then it follows the prototype chain and finds `stomach` in `hamster`.
4. Then it calls `push` on it, adding the food into *the stomach of the prototype*.

So all hamsters share a single stomach!

Every time the `stomach` is taken from the prototype, then `stomach.push` modifies it "at place".

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 evade 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, are usually written into that object. That prevents such problems.

To formulation

F.prototype

Changing "prototype"

Answers:

1. true.

The assignment to `Rabbit.prototype` sets up `[[Prototype]]` for new objects, but it does not affect the existing ones.

2. false.

Objects are assigned by reference. The object from `Rabbit.prototype` is not duplicated, it's still a single object is 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.

3. 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.

4. undefined.

The property `eats` is deleted from the prototype, it doesn't exist any more.

[To formulation](#)

Create an object with the same constructor

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 say, overwrites `User.prototype` and forgets to recreate "constructor", 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:

1. First, it looks for `constructor` in `user`. Nothing.
2. Then it follows the prototype chain. The prototype of `user` is `User.prototype`, and it also has nothing.
3. The value of `User.prototype` is a plain object `{}`, its prototype is `Object.prototype`. And there is `Object.prototype.constructor == Object`. So it is used.

At the end, we have `let user2 = new Object('Pete')`. The built-in `Object` constructor ignores arguments, it always creates an empty object – that's what we have in `user2` after all.

[To formulation](#)

Native prototypes

Add method "f.defer(ms)" to functions

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
}

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

f.defer(1000); // shows "Hello!" after 1 sec
```

[To formulation](#)

Add the decorating "defer()" to functions

```
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
```

To formulation

Methods for prototypes

Add `toString` to the dictionary

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 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.

To formulation

The difference beteeen calls

The 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
```

To formulation

Class patterns

An error in the inheritance

Here's the line with the error:

```
Rabbit.prototype = Animal.prototype;
```

Here `Rabbit.prototype` and `Animal.prototype` become the same object. So methods of both classes become mixed in that object.

As a result, `Rabbit.prototype.walk` overwrites `Animal.prototype.walk`, so all animals start to bounce:

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.walk = function() {
  alert(this.name + ' walks');
};

function Rabbit(name) {
  this.name = name;
```

```
}
```

```
Rabbit.prototype = Animal.prototype;
```

```
Rabbit.prototype.walk = function() {
  alert(this.name + " bounces!");
};
```

```
let animal = new Animal("pig");
animal.walk(); // pig bounces!
```

The correct variant would be:

```
Rabbit.prototype.__proto__ = Animal.prototype;
// or like this:
Rabbit.prototype = Object.create(Animal.prototype);
```

That makes prototypes separate, each of them stores methods of the corresponding class, but `Rabbit.prototype` inherits from `Animal.prototype`.

[To formulation](#)

Rewrite to prototypes

Please note that properties that were internal in functional style (`template`, `timer`) and the internal method `render` are marked private with the underscore `_`.

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Classes

Rewrite to class

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Class inheritance, super

Error creating an instance

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
```

[To formulation](#)

Extended clock

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Class extends Object?

First, 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 important difference in "class Rabbit extends Object" versus `class Rabbit`.

As we know, the “extends” syntax sets up two prototypes:

1. Between “prototype” of the constructor functions (for methods).
2. Between the constructor functions itself (for static methods).

In our case, for `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 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 {}

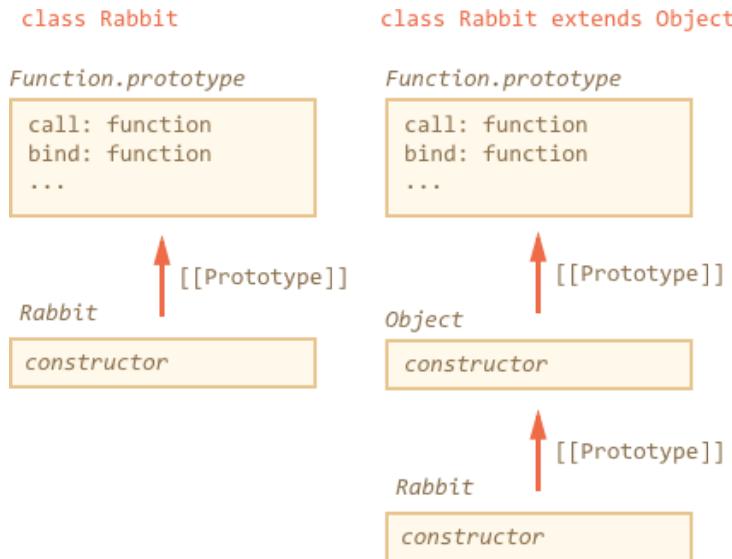
alert( Rabbit.prototype.__proto__ === Object.prototype ); // (1) true
alert( Rabbit.__proto__ === Object ); // (2) false (!)
alert( Rabbit.__proto__ === Function.prototype ); // as any function by default

// 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` 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 <code>super()</code> in constructor
<code>Rabbit.__proto__ === Function.prototype</code>	<code>Rabbit.__proto__ === Object</code>

[To formulation](#)

Class checking: "instanceof"

Strange instanceof

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.

[To formulation](#)

Error handling, "try..catch"

Finally or just the code?

The 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 (e) {
    /// ...
  } finally {
    alert('cleanup!');
  }
}

f(); // cleanup!
```

...Or when there's a `throw`, like here:

```
function f() {
  try {
    alert('start');
    throw new Error("an error");
  } catch (e) {
    // ...
    if("can't handle the error") {
      throw e;
    }
  } 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.

To formulation

Custom errors, extending `Error`

Inherit from `SyntaxError`

```
class FormatError extends SyntaxError {
  constructor(message) {
    super(message);
```

```
    this.name = "FormatError";
}
}

let err = new FormatError("formatting error");

alert( err.message ); // formatting error
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof SyntaxError ); // true
```

To formulation