

Part 1

The JavaScript language

JS

Ilya Kantor

Built at March 7, 2019

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](#)

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 a web page's HTML and executed automatically as the page loads.

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

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

Why JavaScript?

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

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

Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called [the JavaScript engine](#) .

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

Different engines have different “codenames”. For example:

- [V8](#) – in Chrome 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.

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 at each step 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. When it’s done, scripts run quite fast.



What can in-browser JavaScript do?

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.

Javascript’s capabilities greatly depend on the environment it’s running in. For instance, [Node.JS](#)  supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc.

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

For instance, in-browser JavaScript is able to:

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

What CAN’T in-browser JavaScript do?

JavaScript's abilities in the browser are limited 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.

Examples of such restrictions include:

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

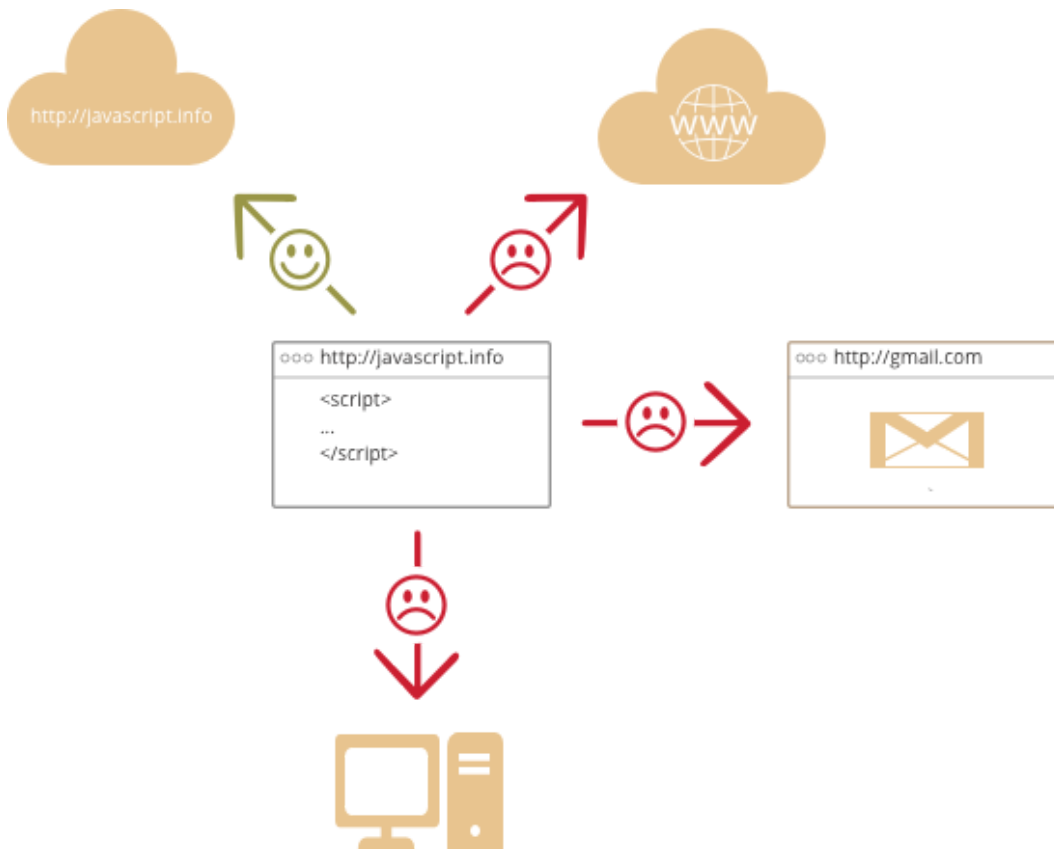
There are ways to interact with camera/microphone and other devices, but they require a user's explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the [NSA](#) .

- 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.

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

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



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

What makes JavaScript unique?

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

- Full integration with HTML/CSS.
- Simple things are done simply.
- Support by all major browsers and enabled by default.

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

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

While planning to learn a new technology, it's beneficial to check its perspectives. So let's move on to the modern trends affecting it, including 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 us to write clearer and more precise code. Usually, Ruby devs like it.
- [TypeScript](#)  is concentrated on adding “strict data typing” to simplify the development and support of complex systems. It is developed by Microsoft.
- [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 these languages, we should also know JavaScript to really understand what we’re doing.

Summary

- JavaScript was initially created as a browser-only language, but is now used in many other environments as well.
- Today, 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 main types of code editors: IDEs and lightweight editors. Many people use one tool of each type.

IDE

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

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

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

- [WebStorm](#) for frontend development. The same company offers other editors for other languages (paid).
- [Netbeans](#) (free).

All of these IDEs are cross-platform.

For Windows, there’s also “Visual Studio”, not to be confused with “Visual Studio Code.” “Visual Studio” is a paid and mighty Windows-only editor, well-suited for the .NET platform. 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) also has many IDE-like features.
- [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:

- As an IDE for JS – [WebStorm](#) (I switch to one of the other JetBrains offerings when using other languages)
- 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, and personal preferences.

Developer console

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

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

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

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

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

Google Chrome

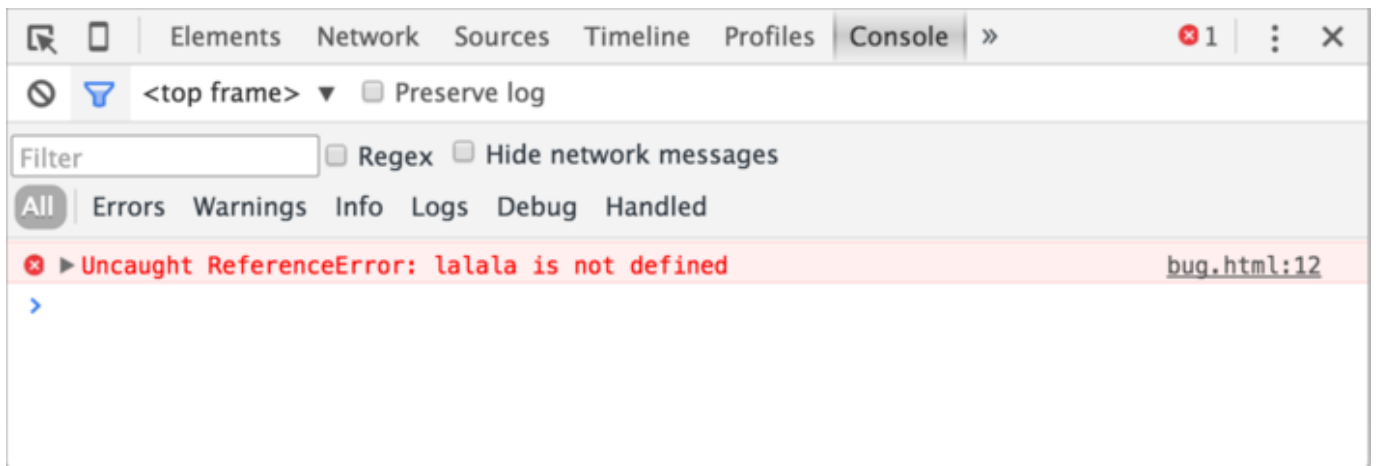
Open the page [bug.html](#).

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 come back to developer tools later and cover debugging more in-depth in the chapter [Debugging in Chrome](#).

Firefox, Edge, and others

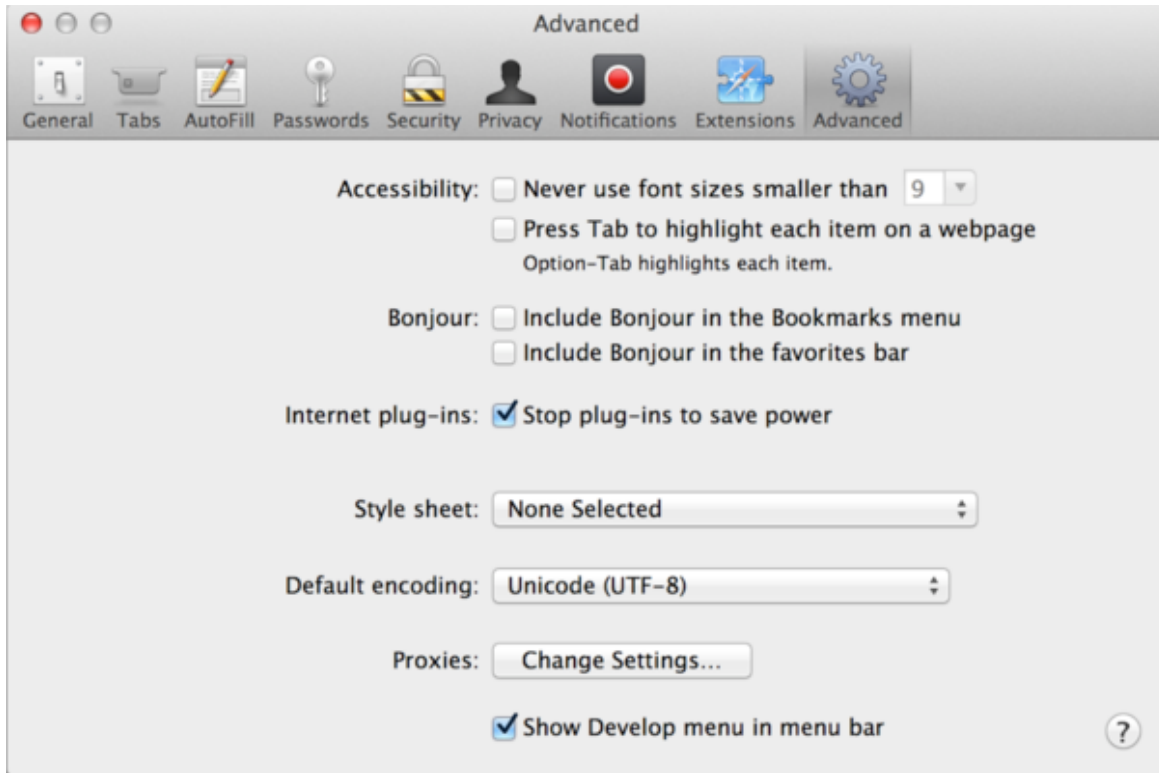
Most other browsers use `F12` to open developer tools.

The look & feel of them is quite similar. Once you know how to use one of these 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 the “Advanced” pane. There's a checkbox at the bottom:



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

Multi-line input

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

To insert multiple line, press `Shift+Enter`.

Summary

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

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

