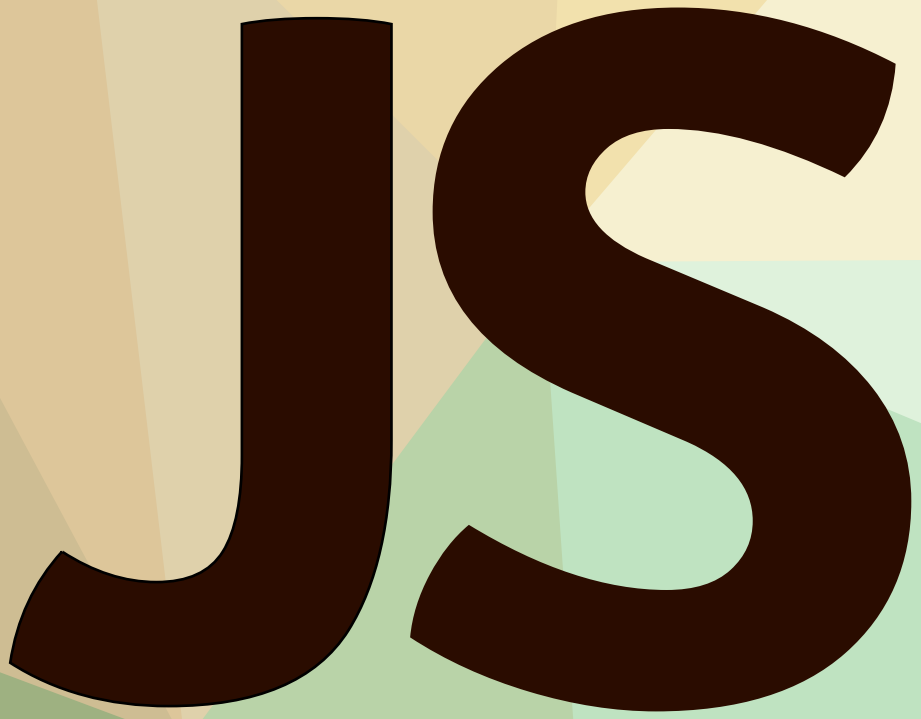
Part 1

The JavaScript language



Ilya Kantor

Built at May 19, 2024

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.

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 what other technologies play well with it.

What is JavaScript?

JavaScript was initially created to "make web pages alive".

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

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

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

> **Why is it called JavaScript?**
>
> When JavaScript was created, it initially had another name: "LiveScript". But Java was very popular at that time, so it was decided that positioning a new language as a "younger brother" of Java would help.
>
> But as it evolved, JavaScript became a fully independent language with its own specification called ECMAScript , and now it has no relation to Java at all.

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

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

Different engines have different "codenames". For example:

- V8 – in Chrome, Opera and Edge.
- SpiderMonkey – in Firefox.
- …There are other codenames like "Chakra" for IE, "JavaScriptCore", "Nitro" and "SquirrelFish" for Safari, etc.

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

How do engines work?

Engines are complicated. But the basics are easy.

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

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

What can in-browser JavaScript do?

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

JavaScript's capabilities greatly depend on the environment it's running in. For instance, Node.js supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc.

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

For instance, in-browser JavaScript is able to:

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

What CAN'T in-browser JavaScript do?

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

Examples of such restrictions include:

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

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

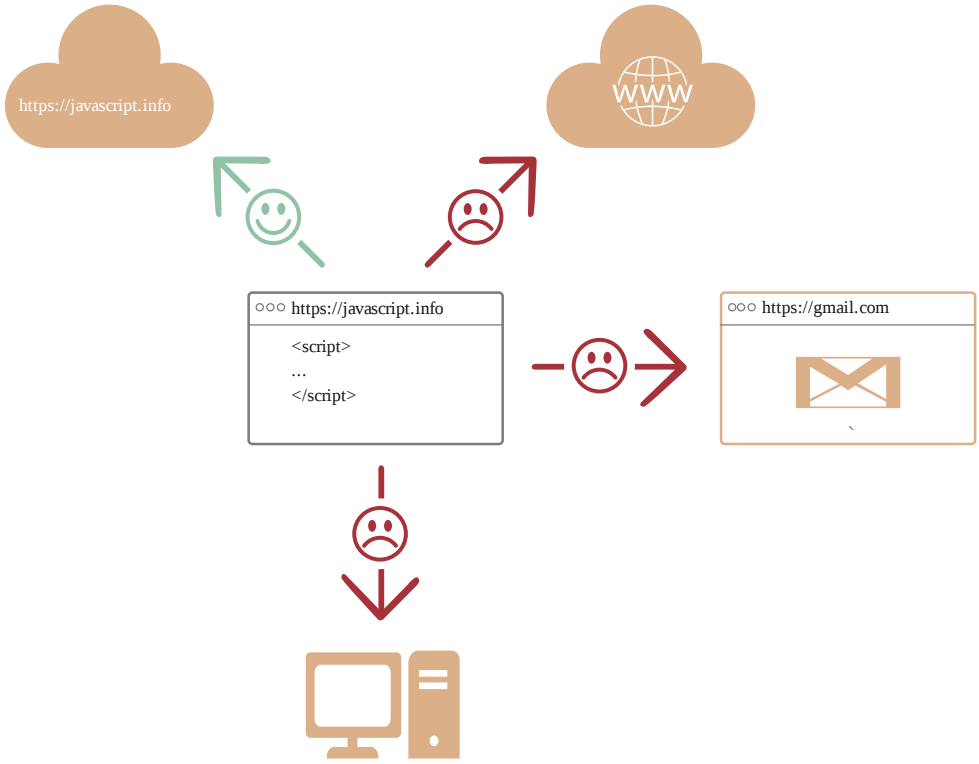
  There are ways to interact with the camera/microphone and other devices, but they require a user's explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the NSA .

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

  This is called the “Same Origin Policy”. To work around that, *both pages* must agree for data exchange and must contain special JavaScript code that handles it. We’ll cover that in the tutorial.

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

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



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

What makes JavaScript unique?

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

- Full integration with HTML/CSS.
- Simple things are done simply.

- Supported by all major browsers and enabled by default.

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

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

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

Languages "over" JavaScript

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

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

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

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

Examples of such languages:

- CoffeeScript is "syntactic sugar" for JavaScript. It introduces shorter syntax, allowing us to write clearer and more precise code. Usually, Ruby devs like it.
- TypeScript is concentrated on adding "strict data typing" to simplify the development and support of complex systems. It is developed by Microsoft.
- Flow also adds data typing, but in a different way. Developed by Facebook.
- Dart is a standalone language that has its own engine that runs in non-browser environments (like mobile apps), but also can be transpiled to JavaScript. Developed by Google.
- Brython is a Python transpiler to JavaScript that enables the writing of applications in pure Python without JavaScript.
- Kotlin is a modern, concise and safe programming language that can target the browser or Node.

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

Summary

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

JavaScript.

Manuals and specifications

This book is a *tutorial*. It aims to help you gradually learn the language. But once you're familiar with the basics, you'll need other resources.

Specification

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

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

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

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

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

Manuals

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

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

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

Compatibility tables

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

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

- https://caniuse.com – per-feature tables of support, e.g. to see which engines support modern cryptography functions: https://caniuse.com/#feat=cryptography .
- https://kangax.github.io/compat-table – a table with language features and engines that support those or don't support.

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

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

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:

- Visual Studio Code (cross-platform, free).
- WebStorm (cross-platform, paid).

For Windows, there’s also “Visual Studio”, not to be confused with “Visual Studio Code”. “Visual Studio” is a paid and mighty Windows-only editor, well-suited for the .NET platform. It’s also good at JavaScript. There’s also a free version Visual Studio Community.

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

Lightweight editors

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

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

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

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

There are many options, for instance:

- Sublime Text (cross-platform, shareware).
- Notepad++ (Windows, free).

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

Let's not 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.

The author's personal opinion:

- I'd use Visual Studio Code if I develop mostly frontend.
- Otherwise, if it's mostly another language/platform and partially frontend, then consider other editors, such as XCode (Mac), Visual Studio (Windows) or Jetbrains family (Webstorm, PHPStorm, RubyMine etc, depending on the language).

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

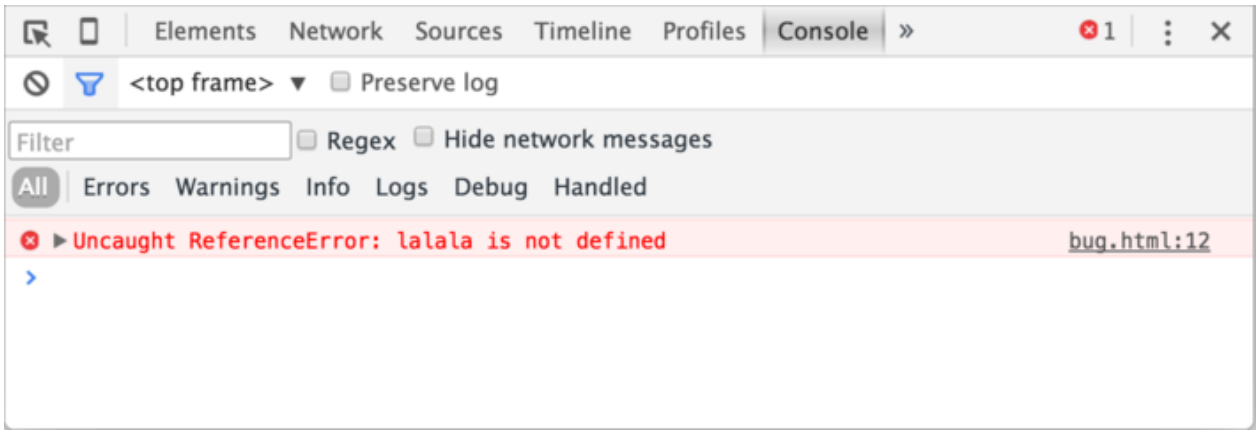
Open the page bug.html.

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.

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

Multi-line input

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

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

Firefox, Edge, and others

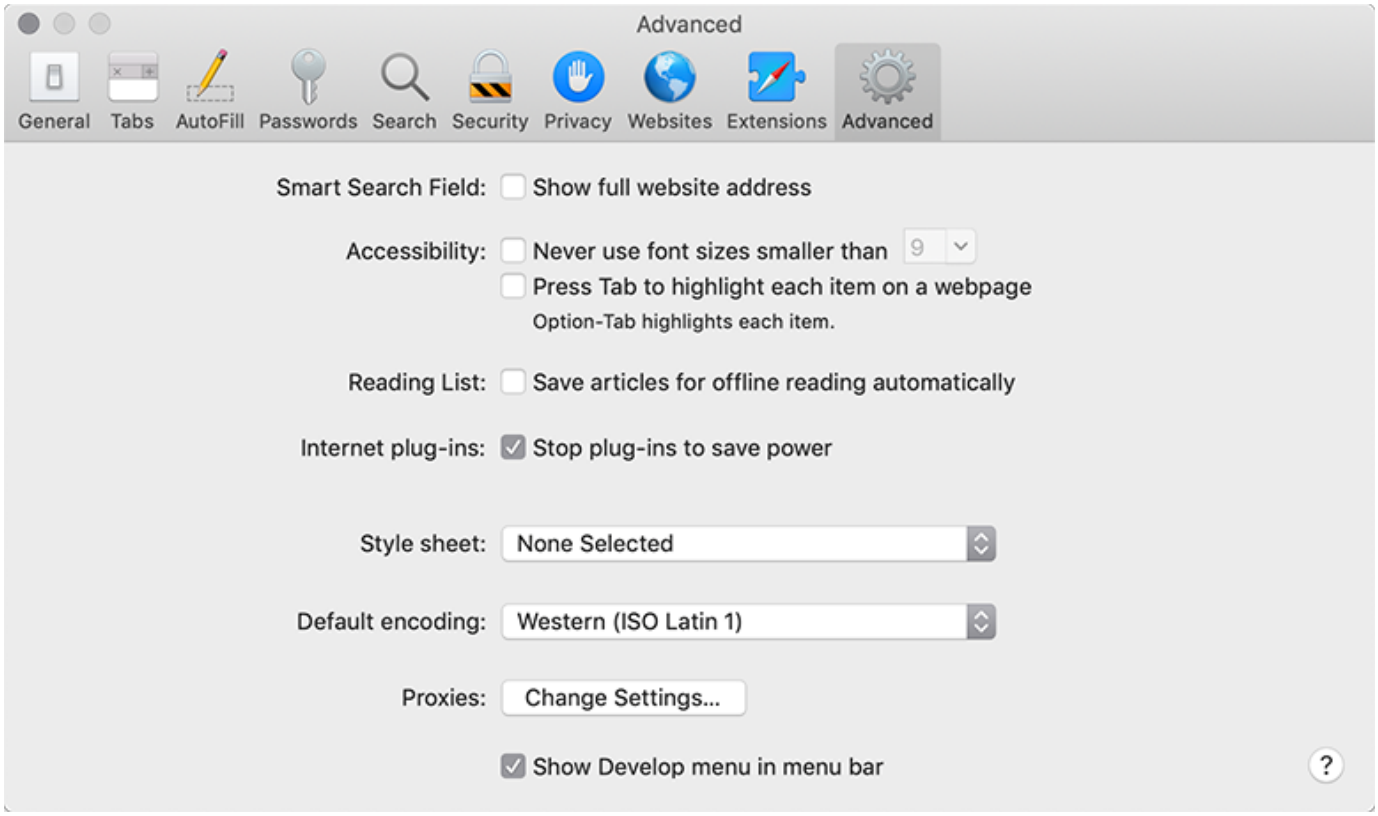
Most other browsers use `F12` to open developer tools.

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.

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.