

**CHAPTER 1**

**Lesson 1: The JavaScript Language**

**TOPICS**

1.1 An Introduction to JavaScript

1.2 Manuals and specifications

1.3 Code editors

1.4 Developer console

**LEARNING OUTCOMES**

At the end of the lesson, you should be able to:

1. understand the basics of Javascript and how it works;
2. construct simple HTML page with the use of Javascript codes;
3. identify the uses of the objects and different elements in Javascript and apply in the development of a web.



**TOPIC 1: AN INTRODUCTION TO JavaScript**

**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.

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

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

# Different engines have different “codenames”. For example:

# V8 – in Chrome, Opera and Edge.

# SpiderMonkey – in Firefox.

# …There are other codenames like “Chakra” for IE, “JavaScriptCore”, “Nitro” and “SquirrelFish” for Safari, etc.

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

# **How do engines work?**

# Engines are complicated. But the basics are easy.

# The engine (embedded if it’s a browser) reads (“parses”) the script.

# Then it converts (“compiles”) the script to machine code.

# And then the machine code runs, pretty fast.

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

# **What can in-browser JavaScript do?**

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

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

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

# For instance, in-browser JavaScript is able to:

# Add new HTML to the page, change the existing content, modify styles.

# React to user actions, run on mouse clicks, pointer movements, key presses.

# Send requests over the network to remote servers, download and upload files (so-called AJAX and COMET technologies).

# Get and set cookies, ask questions to the visitor, show messages.

# Remember the data on the client-side (“local storage”).

# What CAN’T in-browser JavaScript do?

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

# Examples of such restrictions include:

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

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

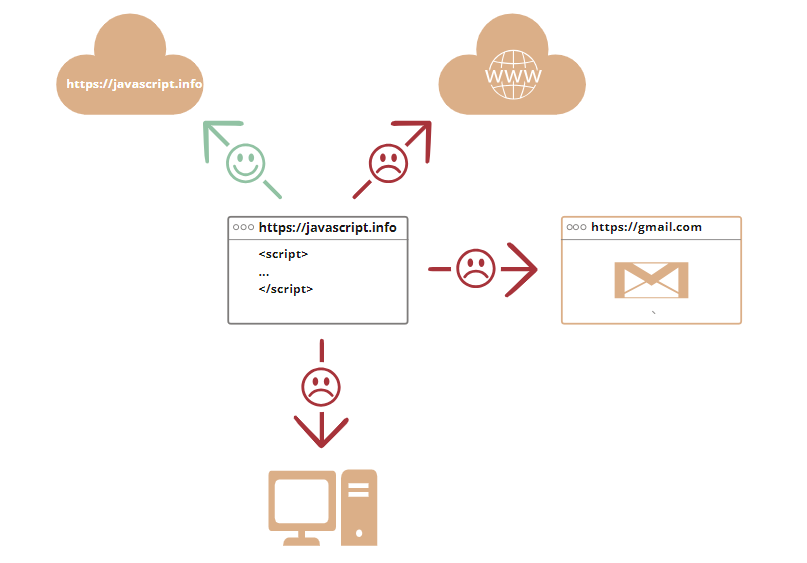
# There are ways to interact with the camera/microphone and other devices, but they require a user’s explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the NSA.

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

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

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

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



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

**What makes JavaScript unique?**

There are at least three great things about JavaScript:

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

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

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

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

**Languages “over” JavaScript**

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

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

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

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

Examples of such languages:

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

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

**Summary**

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



**TOPIC 2: Manuals and specifications**

**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.



**TOPIC 3: Code editors**

**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).



**TOPIC 4: Developer console**

**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**

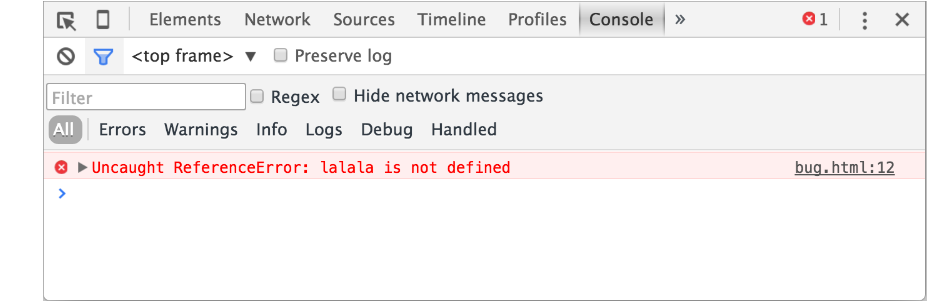
Open the page bug.html.

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:



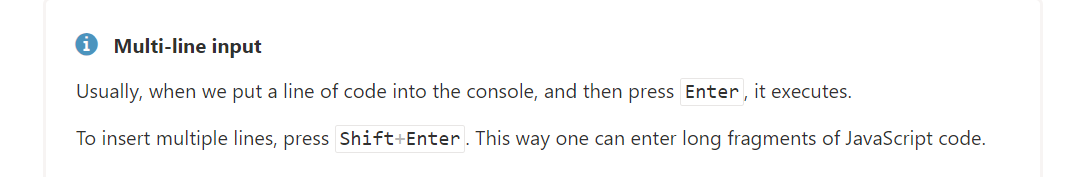
The exact look of developer tools depends on your version of Chrome. It changes from time to time but should be similar.

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.



**Firefox, Edge, and others**

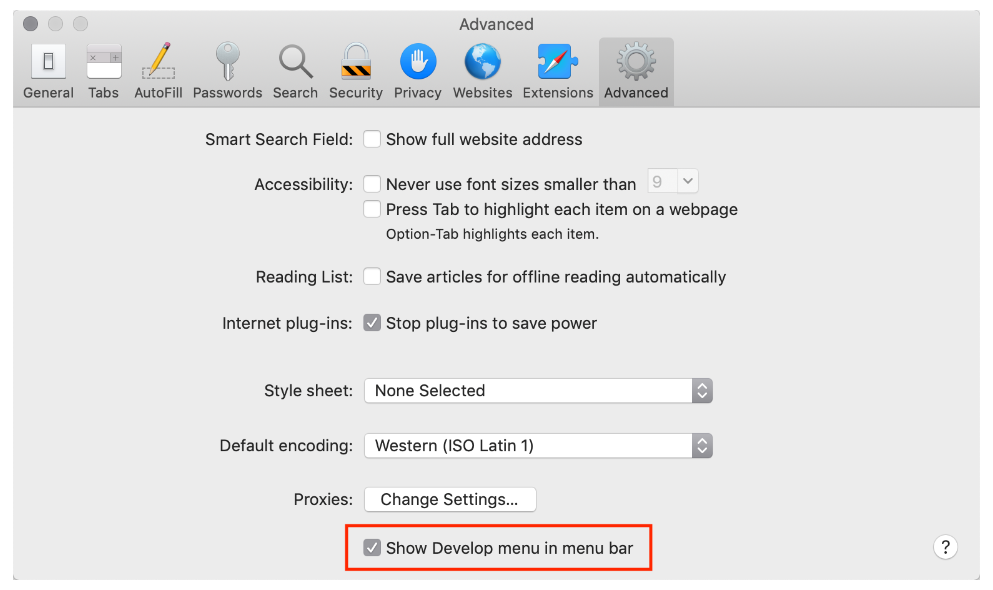
Most other browsers use F12 to open developer tools.

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.



**TOPIC 1: Hello, world!**

**CHAPTER 1**

**Lesson 2: JavaScript Fundamentals**

**TOPICS**

2.1 Hello, world!

2.2 Code structure

2.3 Variables

2.4 Data types

2.5 Basic operators, maths

2.6 Comparison

2.7 Conditional branching: if, ‘?’

2.8 Logical operators

2.9 Loops: while and for

2.10 The “switch” statement

**LEARNING OUTCOMES**

At the end of the lesson, you should be able to:

1. understand the basics of Javascript and how it works;
2. construct simple HTML page with the use of Javascript codes;
3. identify the uses of the objects and different elements in Javascript and apply in the development of a web.

**Hello, world!**

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

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

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

**The “script” tag**

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

For instance:



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

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

**Modern markup**

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

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

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

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

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

**Comments before and after scripts.**

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



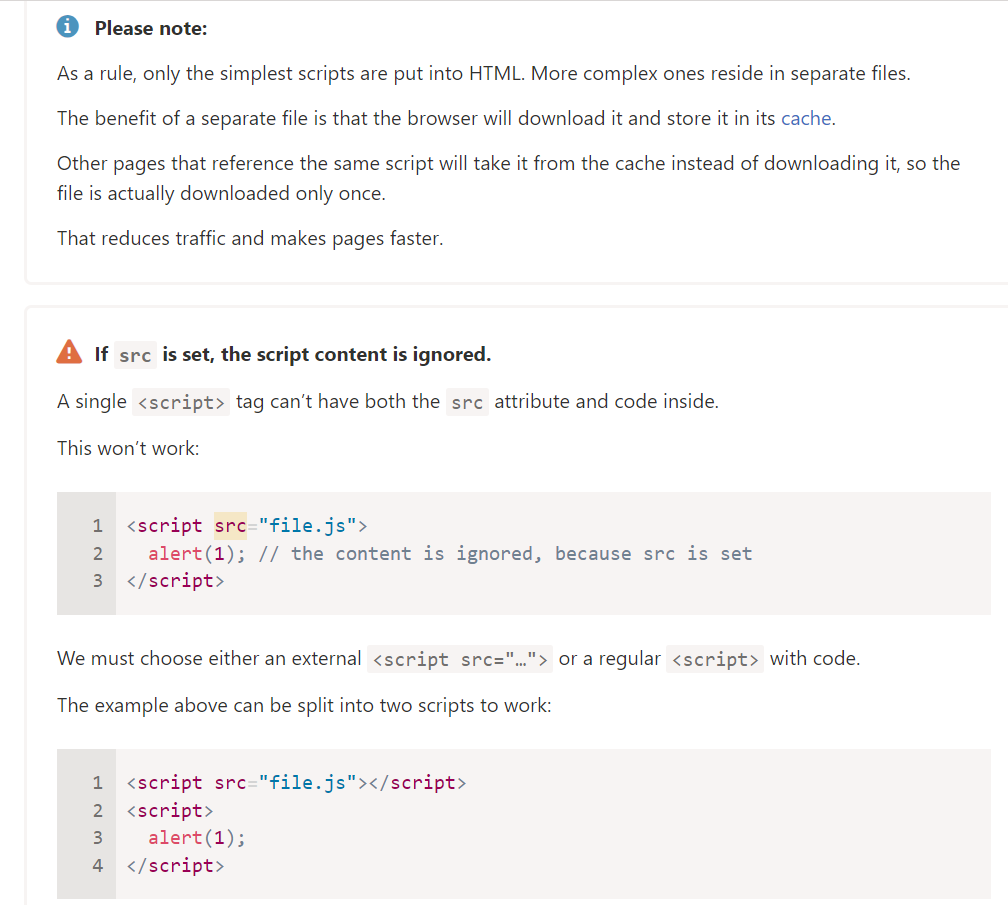
This trick isn’t used in modern JavaScript. These comments hide JavaScript code from old browsers that didn’t know how to process the **<script>** tag. Since browsers released in the last 15 years don’t have this issue, this kind of comment can help you identify really old code.

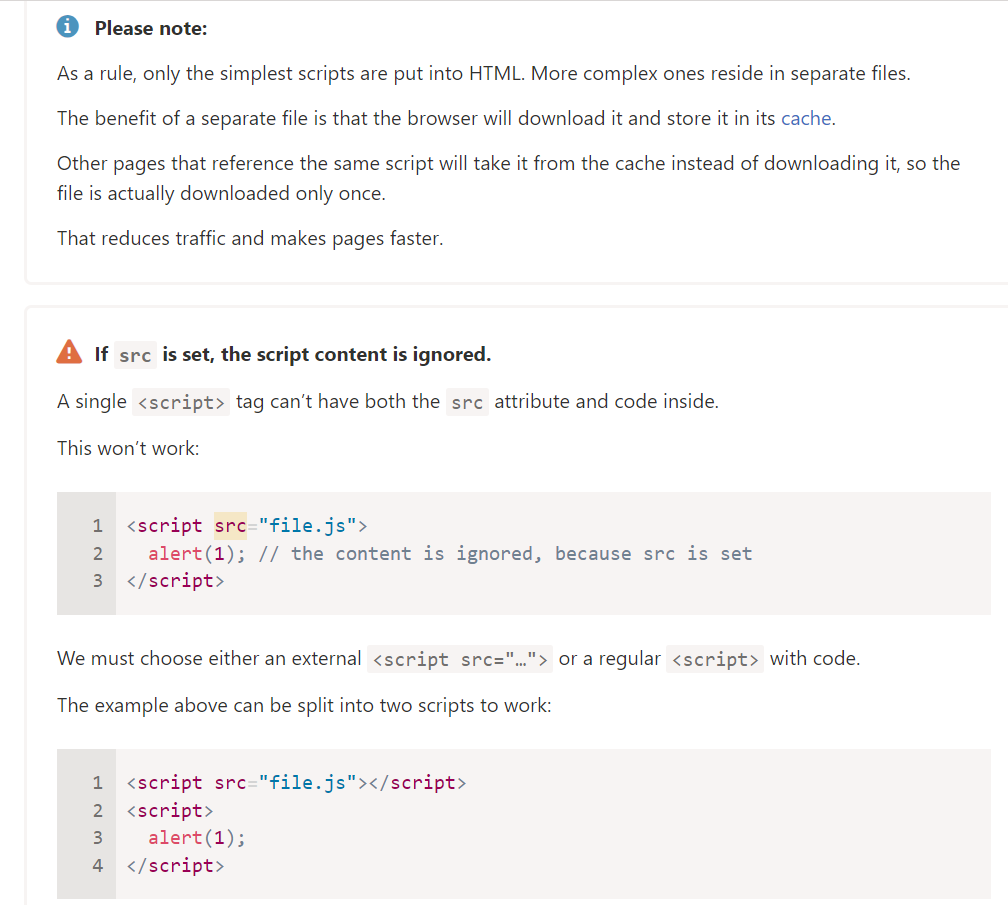
**External scripts**

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

Script files are attached to HTML with the src attribute:







**Summary**

We can use a <script> tag to add JavaScript code to a page.

The **type** and **language** attributes are not required.

A script in an external file can be inserted with **<script src="path/to/script.js"></script>.**

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



**TOPIC 2: Code structure**

**Code structure**

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

**Statements**

Statements are syntax constructs and commands that perform actions.

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

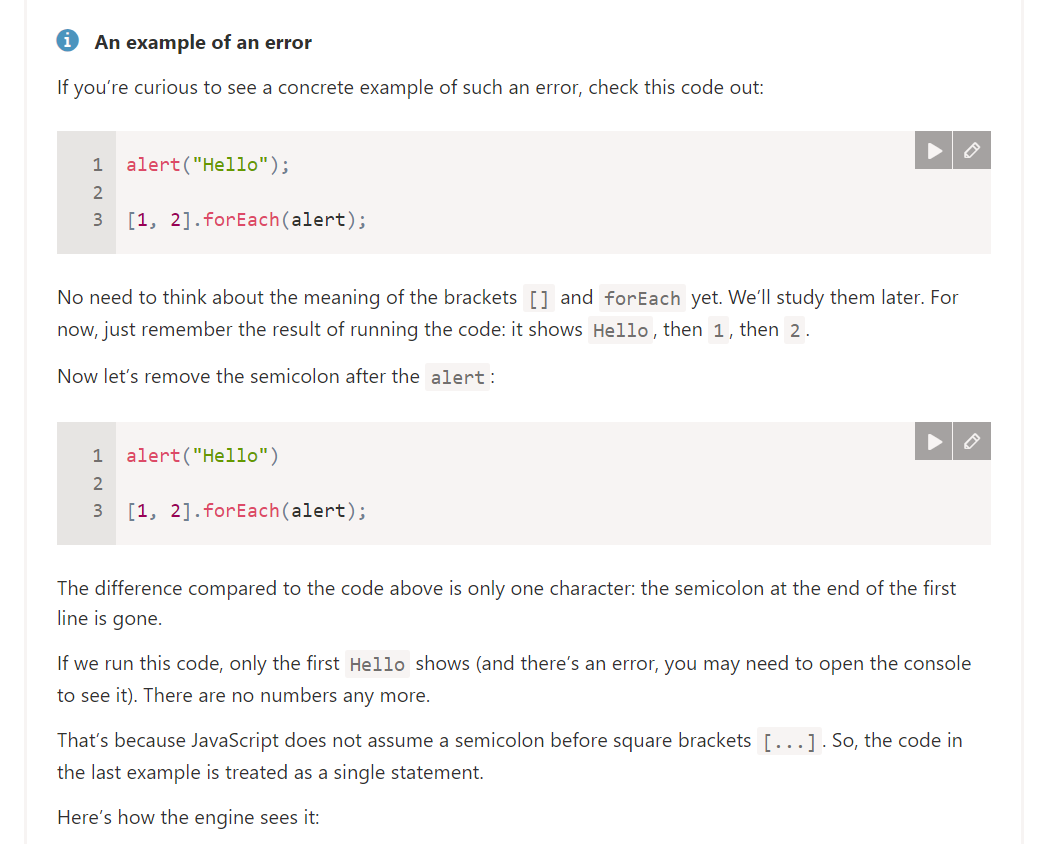
We can have as many statements in our code as we want. Statements can be separated with a semicolon.

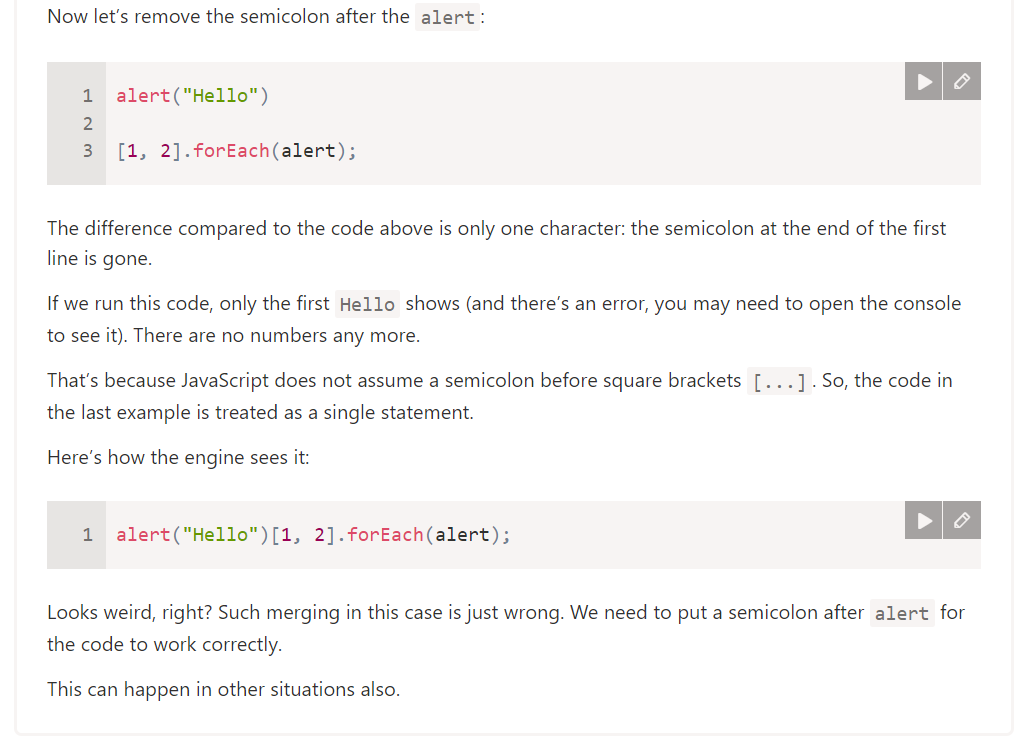
For example, here we split “Hello World” into two alerts:



The code outputs 6 because JavaScript does not insert semicolons here. It is intuitively obvious that if the line ends with a plus "+", then it is an “incomplete expression”, so a semicolon there would be incorrect. And in this case, that works as intended.

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

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



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

**Comments**

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

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

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

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

Like here:







**TOPIC 3: Variables**

**Variables**

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

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

Variables are used to store this information.

**A variable**

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

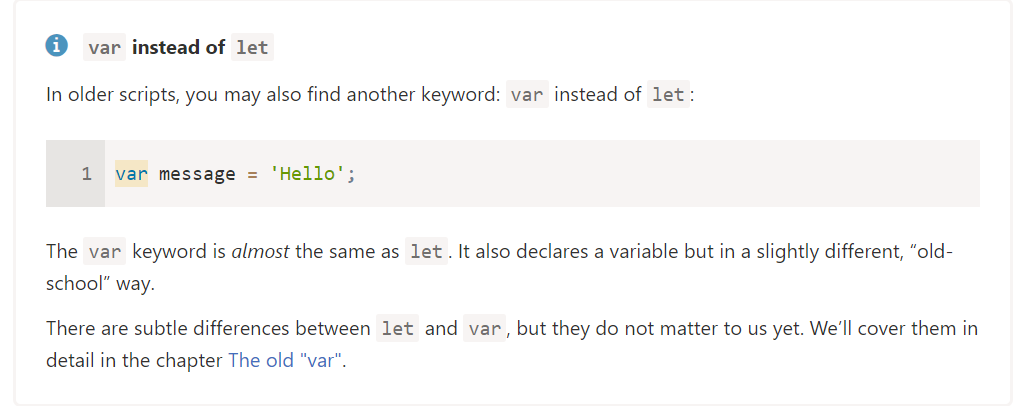
To create a variable in JavaScript, use the let keyword.

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

That might seem shorter, but we don’t recommend it. For the sake of better readability, please use a single line per variable.

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





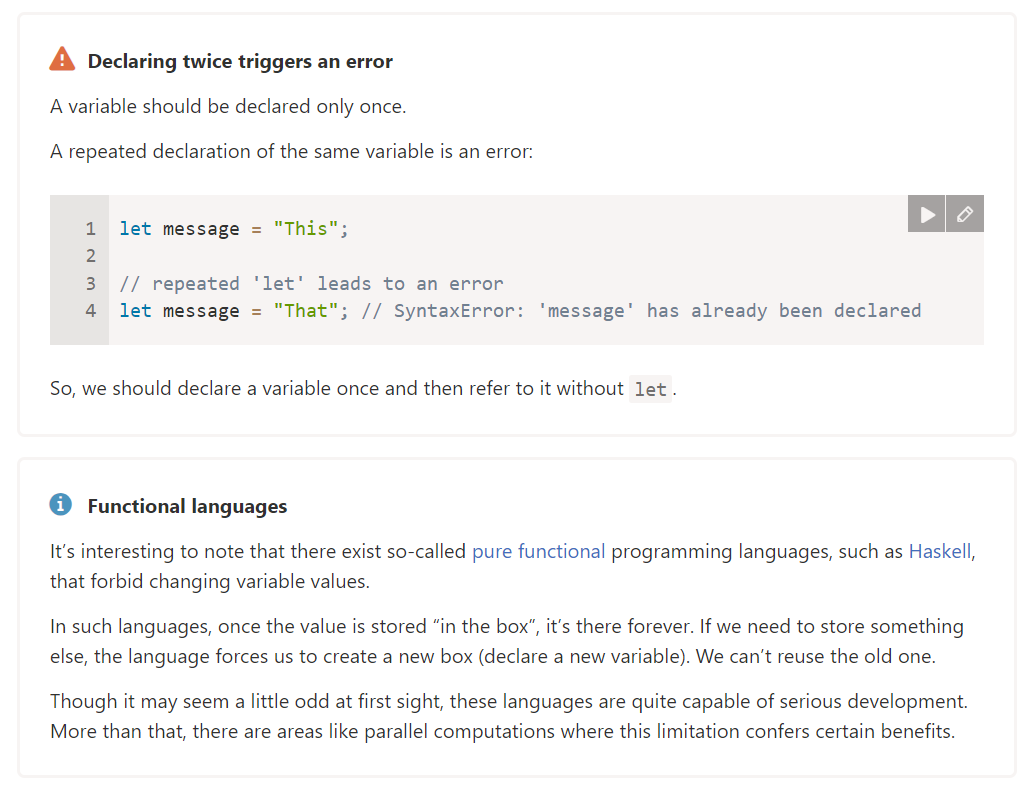
**A real-life analogy**

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 labelled "message" with the value "Hello!" in it:







**Variable naming**

There are two limitations on variable names in JavaScript:

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

Examples of valid names:

let userName;

let test123;

When the name contains multiple words, camelCase is commonly used. That is: words go one after another, each word except first starting with a capital letter: **myVeryLongName.**

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

These names are valid:

let $ = 1; // declared a variable with the name "$"

let \_ = 2; // and now a variable with the name "\_"

alert($ + \_); // 3

Examples of incorrect variable names:

let 1a; // cannot start with a digit

let my-name; // hyphens '-' aren't allowed in the name

Screenshot 2024-09-22 183950J **Case matters**

Variables named apple and APPLE are two different variables.

**Screenshot 2024-09-22 183950Non-Latin letters are allowed, but not recommended**

It is possible to use any language, including Cyrillic letters, Chinese logograms and so on, like this:

let имя = '...';

let 我 = '...';

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

Screenshot 2024-09-22 184621 **Reserved names**

There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

For example: let, class, return, and function are reserved.

The code below gives a syntax error:

let let = 5; // can't name a variable "let", error!

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

Screenshot 2024-09-22 184621**An assignment without use strict**

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

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

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

alert(num); // 5

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

"use strict";

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

**Constants**

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

const myBirthday = '18.04.1982';

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

const myBirthday = '18.04.1982';

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

When a programmer is sure that a variable will never change, they can declare it with const to guarantee and communicate that fact to everyone.

**Uppercase constants**

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known before execution.

Such constants are named using capital letters and underscores.

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

const COLOR\_RED = "#F00";

const COLOR\_GREEN = "#0F0";

const COLOR\_BLUE = "#00F";

const COLOR\_ORANGE = "#FF7F00";

// ...when we need to pick a color

let color = COLOR\_ORANGE;

alert(color); // #FF7F00

**Benefits:**

COLOR\_ORANGE is much easier to remember than "#FF7F00".

It is much easier to mistype "#FF7F00" than COLOR\_ORANGE.

When reading the code, COLOR\_ORANGE is much more meaningful than #FF7F00.

When should we use capitals for a constant and when should we name it normally? Let’s make that clear.

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

For instance:

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

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

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

**Name things right**

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

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

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

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

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

Some good-to-follow rules are:

* Use human-readable names like userName or shoppingCart.
* Stay away from abbreviations or short names like a, b, and c, unless you know what you’re doing.
* Make names maximally descriptive and concise. Examples of bad names are data and value. Such names say nothing. It’s only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.
* Agree on terms within your team and in your mind. If a site visitor is called a “user” then we should name related variables currentUser or newUser instead of currentVisitor or newManInTown.

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

**Screenshot 2024-09-22 183950Screenshot 2024-09-22 183950Reuse or create?**

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

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

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

An extra variable is good, not evil.

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

**Summary**

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

let – is a modern variable declaration.

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

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

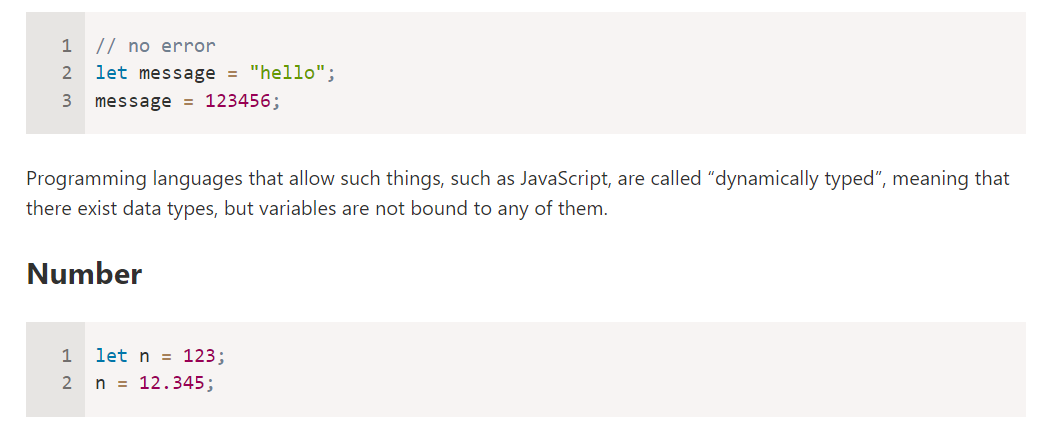
Variables should be named in a way that allows us to easily understand what’s inside them.



**TOPIC 4: Data types**

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

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

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

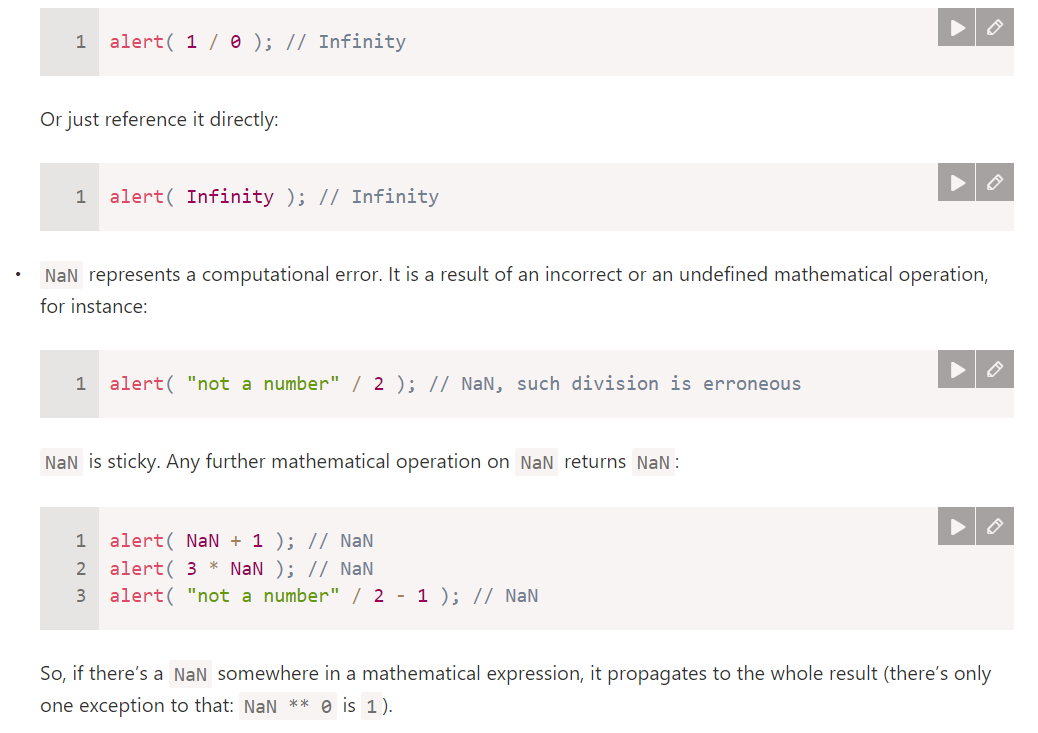
The number type represents both integer and floating point numbers.

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

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

* Infinity represents the mathematical Infinity ∞. It is a special value that’s greater than any number.

We can get it as a result of division by zero:

****

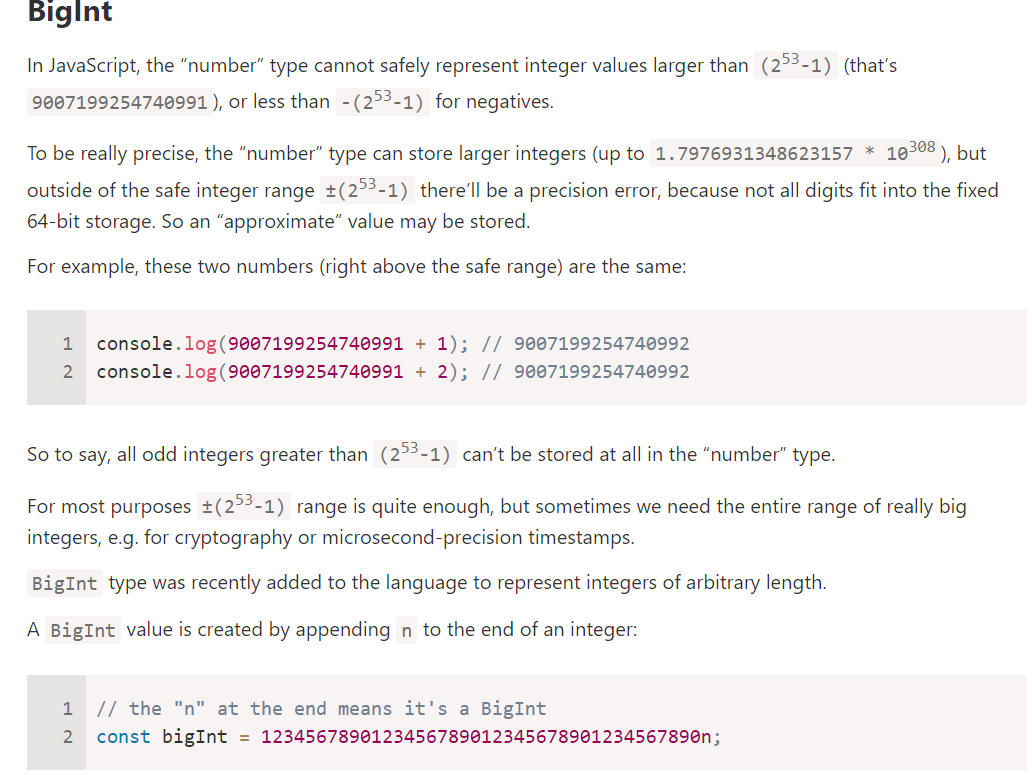
**Screenshot 2024-09-22 183950Mathematical operations are safe**

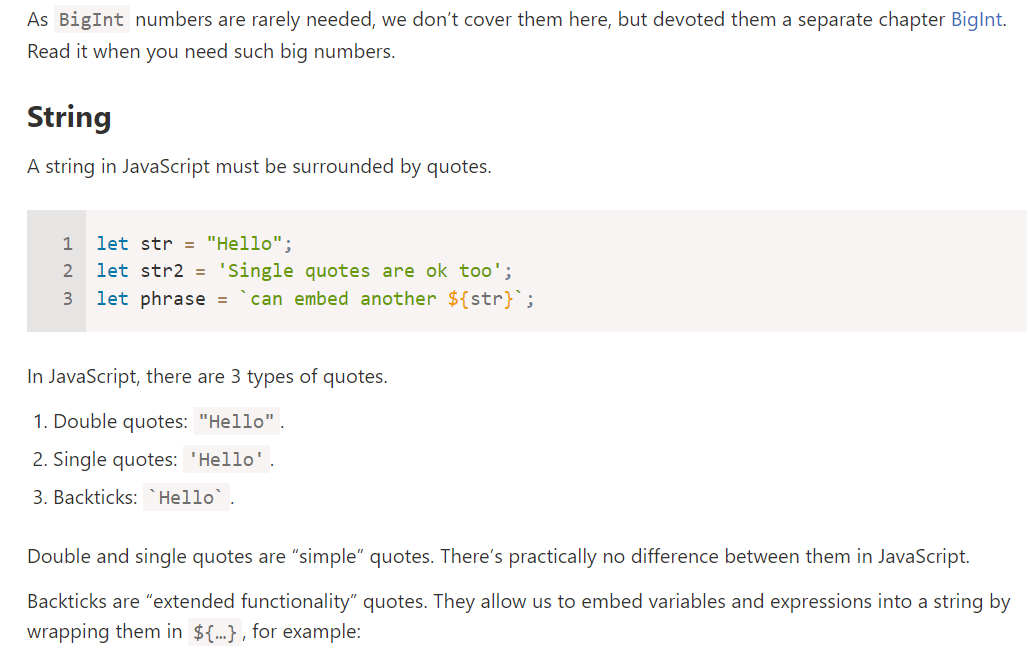
Doing maths is “safe” in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc.

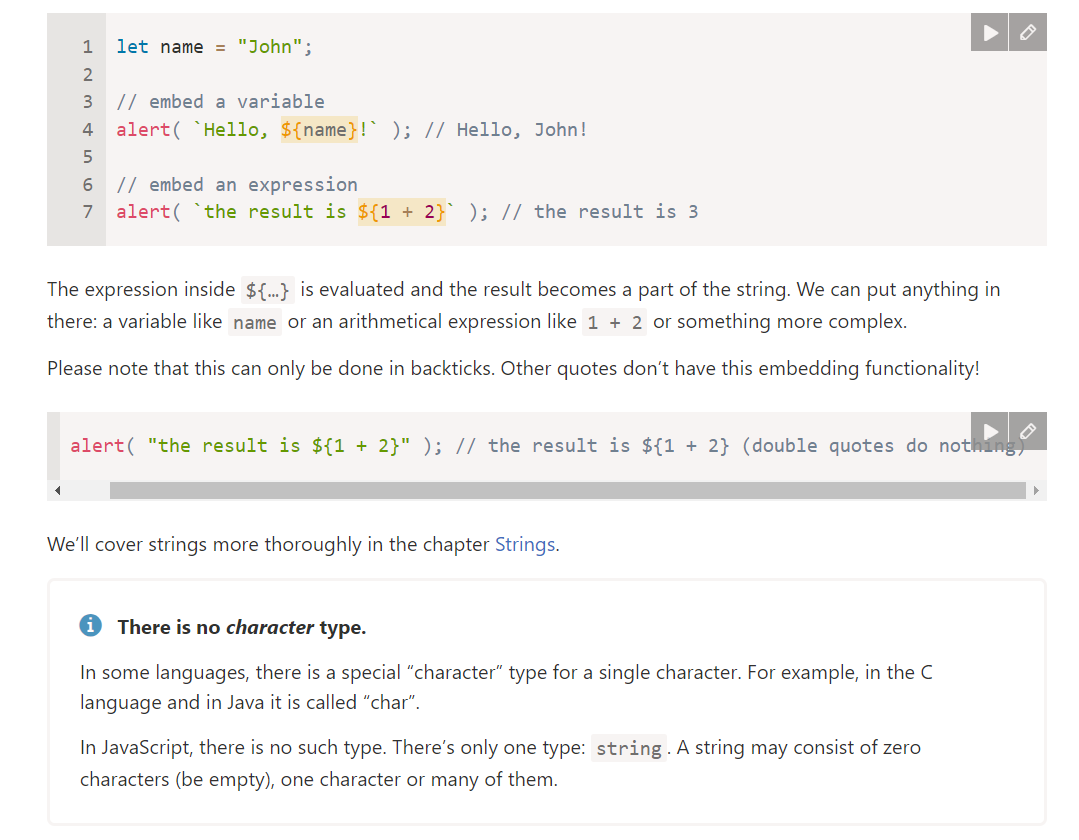
The script will never stop with a fatal error (“die”). At worst, we’ll get NaN as the result.

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

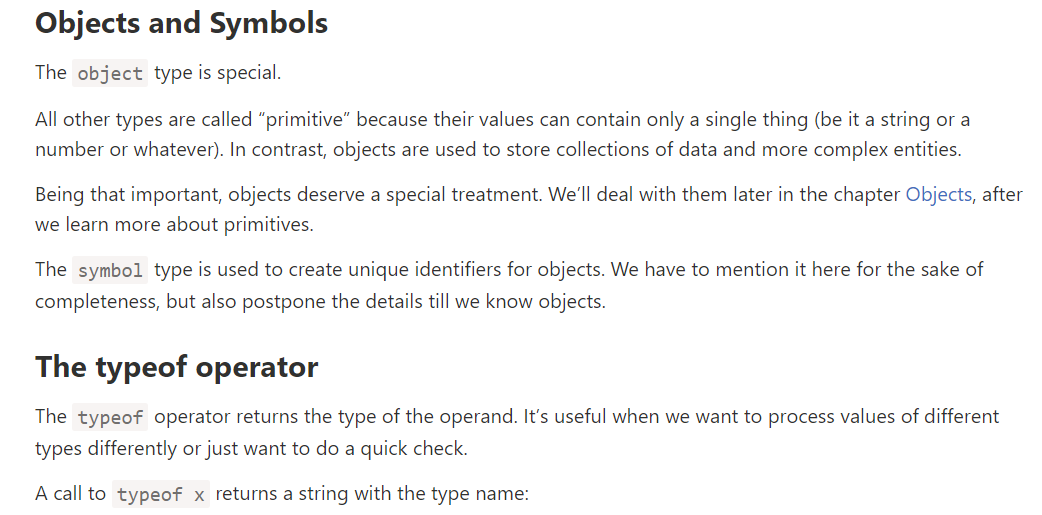
We’ll see more about working with numbers in the chapter Numbers.

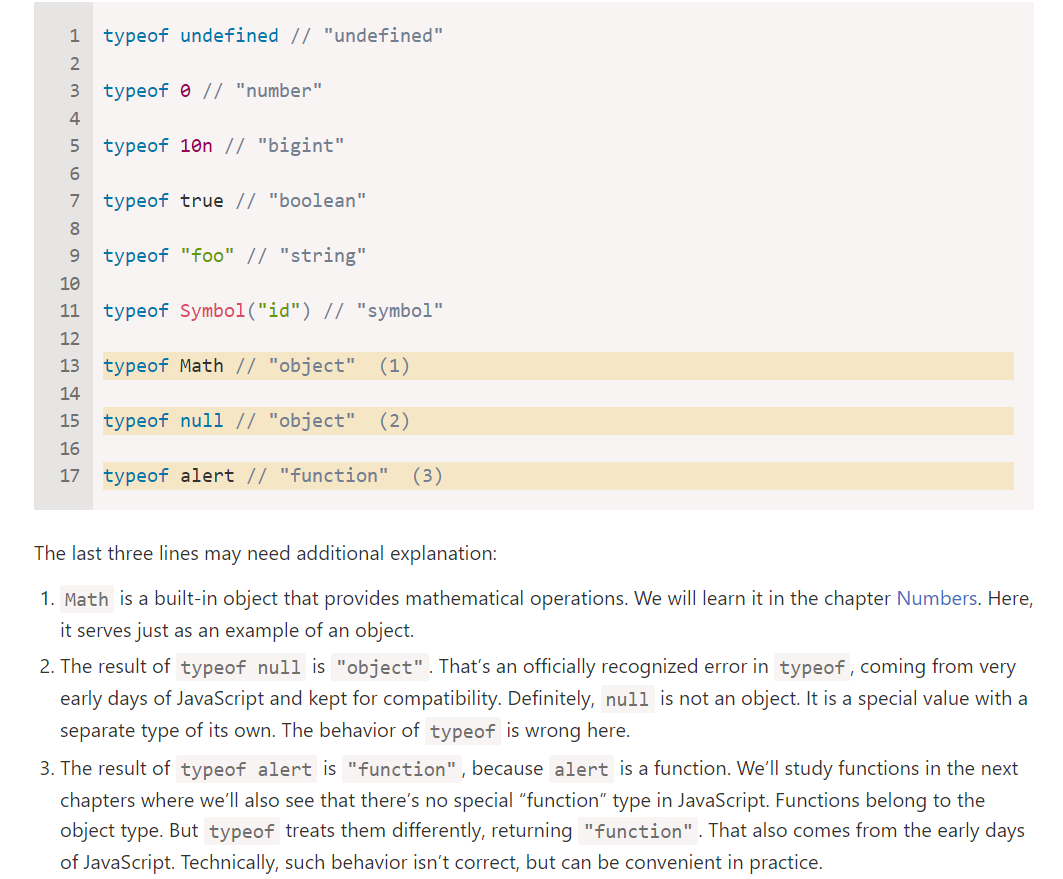


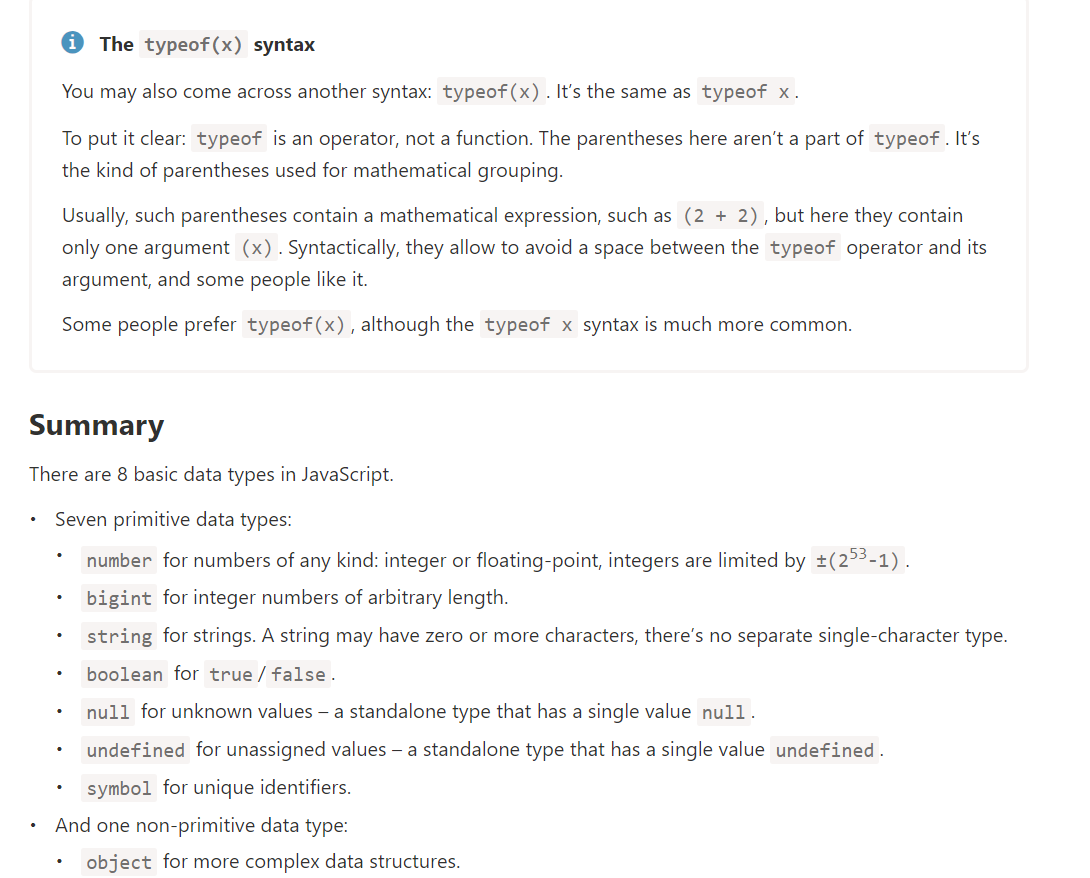


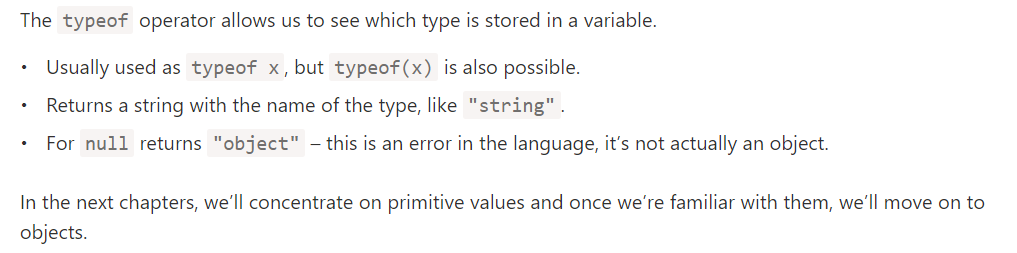














**TOPIC 5: Basic operators, maths**

# **Basic operators, maths**

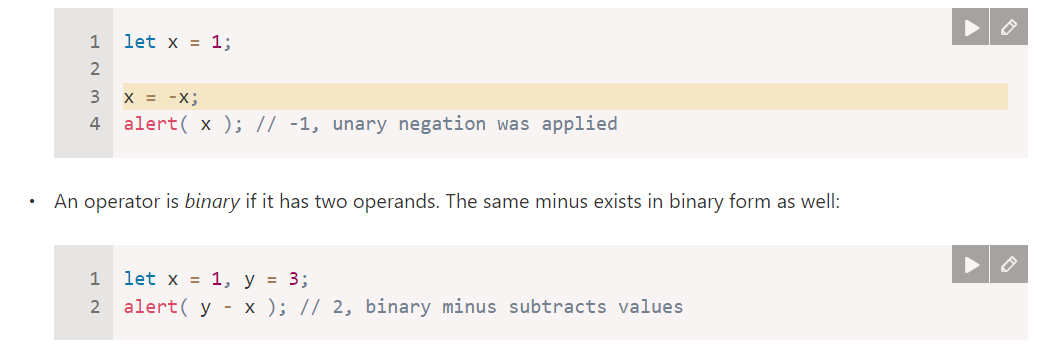
We know many operators from school. They are things like addition +, multiplication \*, subtraction -, and so on.

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

## [Terms: “unary”, “binary”, “operand”](https://javascript.info/operators" \l "terms-unary-binary-operand)

Before we move on, let’s grasp some common terminology.

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



Formally, in the examples above we have two different operators that share the same symbol: the negation operator, a unary operator that reverses the sign, and the subtraction operator, a binary operator that subtracts one number from another.

**Maths**

The following math operations are supported:

Addition +,

Subtraction -,

Multiplication \*,

Division /,

Remainder %,

Exponentiation \*\*.

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

**Remainder %**

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

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

For instance:

