JSE 1

[JavaScript Essentials 1 2](#_Toc159818862)

[JSE: Module 1: Introduction to JavaScript and Computer Programming 3](#_Toc159818863)

[1.0 Welcome to JavaScript Essentials 1 3](#_Toc159818864)

[1.0.1 About the course 3](#_Toc159818865)

[1.0.2 Course Structure 4](#_Toc159818866)

[1.0.3 Syllabus 5](#_Toc159818867)

[1.0.4 Objectives 7](#_Toc159818868)

[1.0.5 Graduate Profile 10](#_Toc159818869)

[1.0.6 Prepare for the JSE-40-0x exam 11](#_Toc159818870)

[1.1 Section 1 – About JavaScript 12](#_Toc159818871)

[1.1.1 Introduction 12](#_Toc159818872)

[1.1.2 How to communicate with the computer 13](#_Toc159818873)

[1.1.3 JavaScript as an interpreted language 14](#_Toc159818874)

[1.1.4 A few more words on JavaScript 15](#_Toc159818875)

[1.1.5 Client-side vs server-side programming 16](#_Toc159818876)

[1.1.6 Is this the perfect programming language? – disadvantages 16](#_Toc159818877)

[1.1.7 Is this the perfect programming language? – advantages 17](#_Toc159818878)

[1.1.8 Let's get ready to work 18](#_Toc159818879)

[1.2 Section 2 – Setting up programming environment 19](#_Toc159818880)

[1.2.1 Development tools 19](#_Toc159818881)

[1.2.2 Online development environment 20](#_Toc159818882)

[1.2.3 Local development environment 21](#_Toc159818883)

[1.2.4 Code editor 22](#_Toc159818884)

[1.2.5 Interpreter 24](#_Toc159818885)

[1.2.6 Debugger 26](#_Toc159818886)

[1.3 Section 3 – Hello, World! 28](#_Toc159818887)

[1.3.1 The "Hello, World!" Program 28](#_Toc159818888)

[1.3.2 Online development environment 30](#_Toc159818889)

[1.3.3 Local development environment 31](#_Toc159818890)

[1.3.4 The script tag 32](#_Toc159818891)

[1.3.5 ... and a little something about CSS 33](#_Toc159818892)

[1.3.6 How can we run our JavaScript code? 34](#_Toc159818893)

[1.3.7 Okay, maybe we could finally run something... 37](#_Toc159818894)

[1.3.8 Executing the code directly in the console 38](#_Toc159818895)

[1.3.9 SECTION SUMMARY 41](#_Toc159818896)

[1.3.10 SECTION QUIZ 41](#_Toc159818897)

[1.4 Module 1 Completion – Module Test 42](#_Toc159818898)

[JSE: Module 2: Variables, Data Types, Type Casting, and Comments 51](#_Toc159818899)

[2.0 Section 1 – Variables 51](#_Toc159818900)

[2.0.1 Variables 51](#_Toc159818901)

[2.0.2 Naming the variables 52](#_Toc159818902)

[2.0.3 Declaring variables 54](#_Toc159818903)

[2.0.4 Initializing variables 56](#_Toc159818904)

[2.0.5 Declarations and strict mode 58](#_Toc159818905)

[2.0.6 Changing variable values 59](#_Toc159818906)

[2.0.7 Constants 60](#_Toc159818907)

# JavaScript Essentials 1

## JSE: Module 1: Introduction to JavaScript and Computer Programming

### 1.0 Welcome to JavaScript Essentials 1

#### 1.0.1 About the course

This course has been designed and developed by [OpenEDG JS Institute](https://js.institute/" \t "_blank), and implemented by the Cisco Networking Academy.



This course is **the first in a 2-course series** that will prepare you for the [JSE – Certified Entry-Level JavaScript Programmer](https://js.institute/jse-certification) and [JSA – Certified Associate JavaScript Programmer](https://js.institute/jsa-certification) certification exams.

The main goal of the course is to guide you from a state of complete programming illiteracy to a level of programming knowledge which allows you to **design, write, debug, and run programs** encoded in the JavaScript language, and to understand the basic concepts of software development technology.

The course focuses on the JavaScript language **core concepts**, including the essentials of the JavaScript syntax and semantics, best programming practices, JS tools and resources, and coding concepts such as variables, data types, type casting, operators, user interaction, control flow, functions, errors, exceptions, debugging, and troubleshooting.

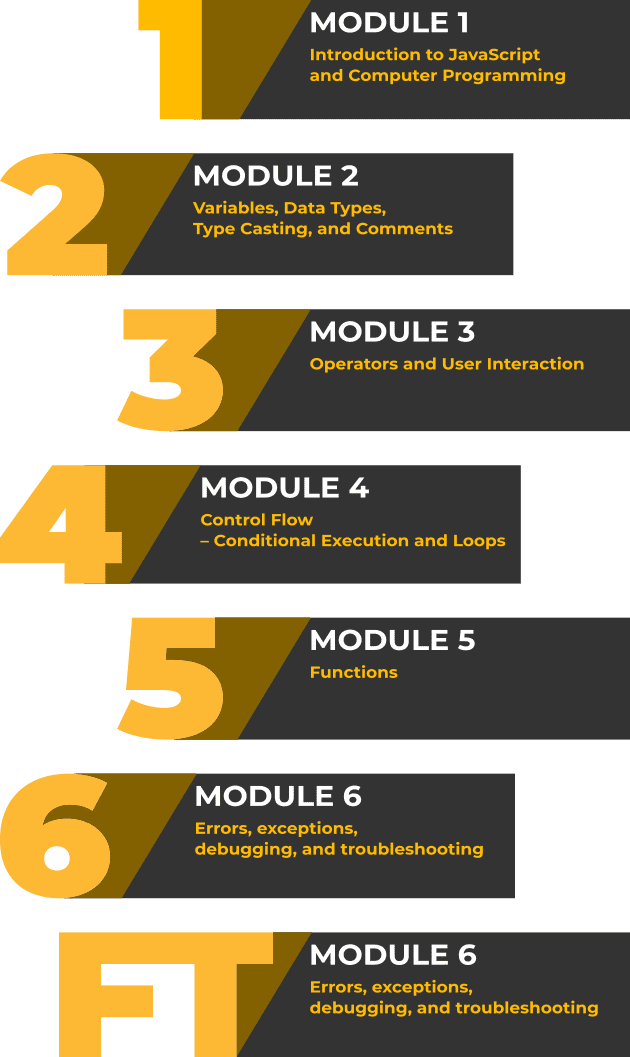
The course will prepare you for jobs and careers connected with widely understood **software and web development**, which includes not only creating the code itself as a junior developer, but also **web design, front-end development**, and **application testing**.

#### 1.0.2 Course Structure

**The course is divided into six modules:**

1. **Module 1  
   Introduction to JavaScript and Computer Programming;**
2. **Module 2  
   Variables, Data Types, Type Casting, and Comments;**
3. **Module 3  
   Operators and User Interaction;**
4. **Module 4  
   Control Flow – Conditional Execution and Loops;**
5. **Module 5  
   Functions;**
6. **Module 6  
   Errors, exceptions, debugging, and troubleshooting.**

**The course concludes with a final test consisting of 30 questions, assessing your comprehension of the concepts covered.**

****

#### 1.0.3 Syllabus

In this course you will learn:

**Module 1**

**Introduction to JavaScript and Computer Programming**

* About JavaScript (how to communicate with the computer, what is JS, advantages and limitations of JS, where is JS used today)
* Setting up the programming environment (development tools, online development environment, local development environment)
* First JS program – Hello, World! (a few words about HTML, how to run your JavaScript code, executing the code directly in the console)

**Module 2**

**Variables, Data Types, Type Casting, and Comments**

* Variables (naming, declaring and initializing variables, declarations and strict mode, changing variable values, constants, scope)
* Primitive data types (Boolean, Number, BigInt, String, undefined, null, type casting – primitive construction functions and primitive conversions, implicit conversions)
* Complex data types (Object, Array, basic Array properties and methods)
* Comments (single-line comments, multi-line comments, documentation)

**Module 3**

**Operators and User Interaction**

* Assignment, arithmetic, and logical operators (what are operators, assignment operators, arithmetic operators, logical operators, compound assignment operators)
* Strings, comparison, and other JS operators (string concatenation and compound assignments, comparison operators, conditional operators, typeof, instanceof and delete operators, operator precedence)
* Interacting with the user (dialog boxes – alert, confirm, prompt)

**Module 4**

**Control Flow – Conditional Execution and Loops**

* Conditional execution (what is conditional execution, the if–else statement, the conditional operator, the switch–case statement)
* Loops (what are loops, the while loop, the do–while loop, the for loop, the for–of loop, the for–in loop, the break and continue statements)

**Module 5**

**Functions**

* Function basics (what are functions, declaring functions, calling functions, local variables, the return statement, function parameters, shadowing)
* Functions as first-class members (function expressions, passing a function as a parameter, callbacks)
* Arrow functions (declaring and calling)
* Recursion (basic idea)

**Module 6**

**Errors, exceptions, debugging, and troubleshooting**

* Errors and exceptions – introduction (natural languages and communication errors, errors vs. exceptions, errors without exceptions, limited confidence)
* Basic types of errors in JS (SyntaxError, ReferenceError, TypeError, RangeError)
* Exception handling (the try–catch statement, the finally statement, the throw statement, and custom errors)
* Code debugging and troubleshooting (what is debugging, step-by-step execution, viewing and modifying variables, the step out option, measuring code execution time)

#### 1.0.4 Objectives

**Module 1**

After completing Module 1, you will:

* understand the fundamental programming concepts, such as: interpreting and the interpreter, compilation and the compiler, client-side vs. server-side programming;
* have basic knowledge of how to set up and use the basic programming environment (online or local)
* gain skills allowing you to run your first JavaScript program on the client side (both as an element embedded in the HTML page and directly in the browser console).

**Module 2**

After completing Module 2, you will:

* have the knowledge and skills to work with variables (i.e. naming, declaring, initializing and modifying their values)
* understand concepts such as scope, code blocks, shadowing, and hoisting;
* know the basic properties of primitive data types such as boolean, number, bigint, undefined, null, and be able to use them;
* be familiar with the basic properties of the primitive data type string, including string literals – single or double quotes, the escape character, string interpolation, basic properties and methods;
* know the basic properties of complex data types such as Array and Object (treated as a record) and be able to use them in practice.

**Module 3**

After completing Module 3, you will:

* know what operators are and how to classify them (by type of operand, by number of operands, etc.)
* be able to use assignment, arithmetic, logical, and comparison operators in practice;
* understand the operation of the conditional operator and the typeof, instanceof, and delete operators;
* understand what the precedence and associativity of basic operators are and be able to influence them by means of bracket grouping;
* be able to perform basic two-way communication with the program user using the alert, confirm, and prompt dialog boxes.

**Module 4**

After completing Module 4, you will:

* be able to force conditional execution of a group of statements (make decisions and branch the flow) using if-else and switch commands;
* be able to force a group of statements to repeat in a loop using the for, while, and do-while commands, using both dependent and independent conditions on the number of iterations;
* understand and be able to use loop-specific break and continue instructions;
* be able to use the for-in statement to iterate over the properties of an object;
* be able to use the for-of statement to walk through the elements of an array.

**Module 5**

After completing Module 5, you will:

* be able to declare and call functions;
* know how to pass call arguments to a function and return the result of its operation from it;
* understand the concept of a local variable and the effect of shadowing variables with the same names within a function;
* know that a function in JS is a first-class member and be able to take advantage of this by declaring functions using function expressions and passing functions as arguments to calls of other functions;
* understand the concept of recursion in the context of functions and be able to solve simple programming problems by using it;
* have a basic understanding of the callback function and be able to use it asynchronously in conjunction with the setTimeout and setInterval methods;
* have a clear understanding of arrow function notation and be able to write alternative functions as regular declarations, function expressions, and arrow functions.

**Module 6**

After completing Module 6, you will:

* gain an understanding of the differences between syntactic, semantic, and logical errors;
* understand the concept of an exception and distinguish between the basic exceptions generated by JS when an error occurs: SyntaxError, ReferenceError, TypeError, RangeError;
* have the ability to handle exceptions using the try-catch-final statement;
* be able to generate your own exceptions using the throw statement;
* have the skills to use the debugger for basic analysis of your own code, including: step-by-step execution, viewing and modifying variables, measuring code execution time.

#### 1.0.5 Graduate Profile

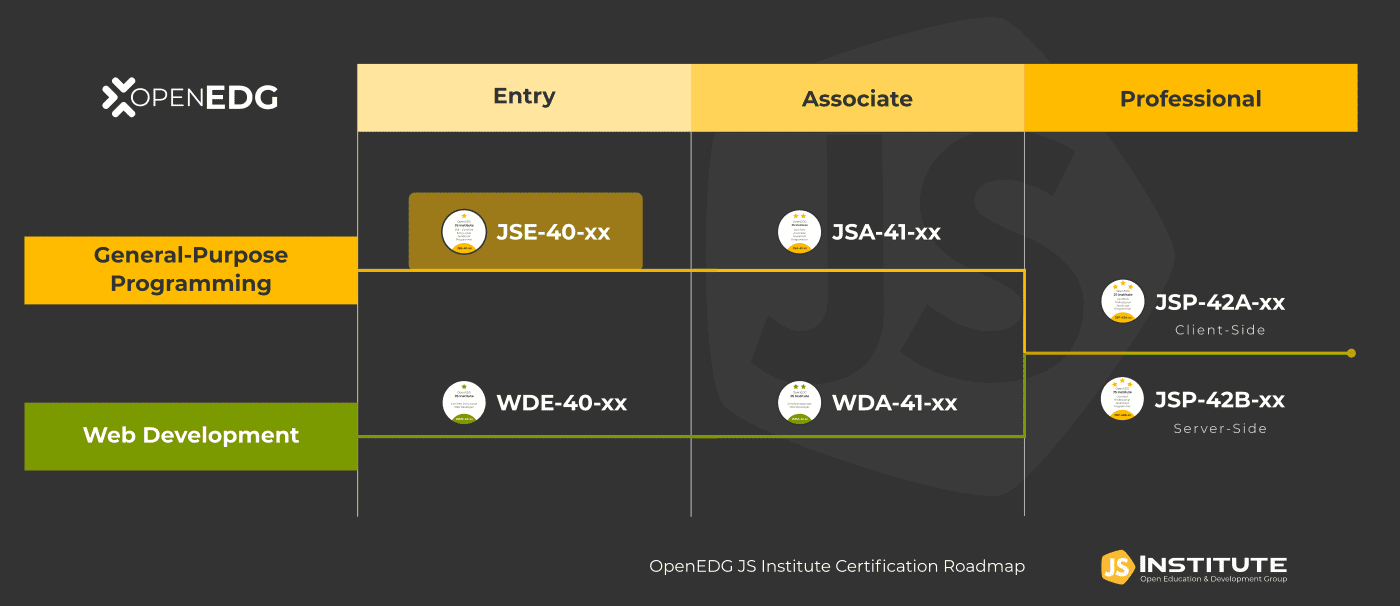
Once you have graduated from the course, you will:

* know the syntax of the core JavaScript language to a degree that will allow for working with variables, operators, flow control, and functions;
* know the basics of the JavaScript data types system, distinguishing between primitive and complex types, and is able to choose a type adequate to their needs;
* think algorithmically and be able to analyze a problem using a programmatic conceptual apparatus;
* be able to choose a data type adequate to the problem being solved and use suitable flow control means;
* be able to design, develop, and improve very simple JavaScript programs;
* be able to interpret and handle basic exceptions related to errors in program execution;
* understand a programmer's work in the software development process and the role of fundamental development tools;
* know how a program is interpreted and executed in an actual computer environment, local or remote;
* have the necessary skills and knowledge to start creating and developing your own programming portfolio.

#### 1.0.6 Prepare for the JSE-40-0x exam

Dive into programming, learn JavaScript from scratch, and prepare for the JSE – Certified Entry-Level JavaScript Programmer certification

* *JavaScript Essentials 1* is directly aligned with the [JSE – Certified Entry-Level JavaScript Programmer](https://js.institute/jse-certification) certification, a professional credential that demonstrates the candidate’s understanding of the JavaScript language core syntax and semantics, as well as their proficiency in using the most essential elements of the language, tools, and resources to design, develop, and refactor simple JavaScript programs.
* The certification holder knows the syntax of the core JavaScript language to a degree that allows them to work with variables, operators, control flow mechanisms, and functions, as well as understands the fundamentals of the JavaScript data type system, exception handling, troubleshooting, debugging, and the runtime environment.



* *JSE – Certified Entry-Level JavaScript Programmer* certification is an interim step to the *JSA – Certified Associate in JavaScript Programming* certification, and the starting point to launch a career in software development, JavaScript programming, and related technologies.
* Becoming JSE certified will help you **stand out** from other candidates and **get your foot in the door**, and after completing this course you'll become eligible for a **20% discount on the certification exam**!
* For more information about the *JSE – Certified Entry-Level JavaScript Programmer* certification, please visit [www.js.institute](http://javascriptinstitute.com/jse-certification).

**So, are you ready to begin your JavaScript journey? Click *Next* to proceed to Section 1 and start learning. See you there!**

### 1.1 Section 1 – About JavaScript

#### 1.1.1 Introduction

So you want to learn how to program in JavaScript? That's great!

This course will be quite a long journey, but if you make it to the end, you will be able to read, understand, and of course, write JavaScript applications and programs. These new abilities may aid you in your current work, or allow you to reach new career opportunities in a steadily growing IT market. Let's start this adventure without any further ado, and let's find out what JavaScript is.

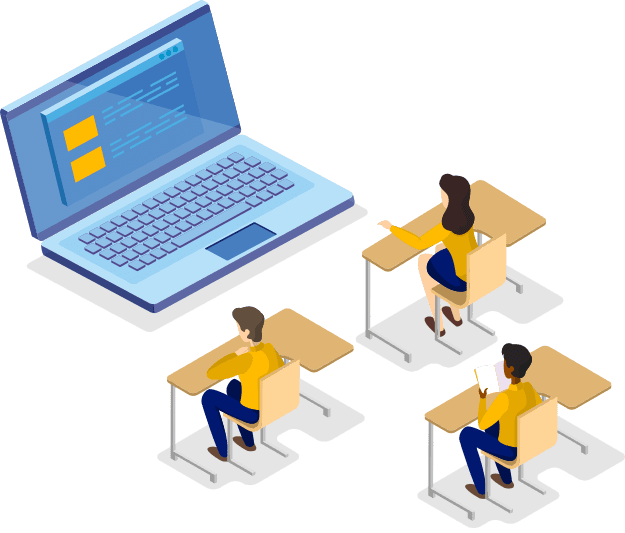
#### 1.1.2 How to communicate with the computer

Let's start with something obvious – computers are everywhere.

You are almost certainly reading this course on a computer; your phone is a computer; you will find computers in TVs and other home appliances, such as your automatic vacuum cleaner, which uses its computer and numerous sensors in its never-ending mission to eradicate dust from your life.

We are surrounded by computers. Computers are used in science, medicine, banking, and business.

We suspect that nowadays it would be difficult to find any area of life in which computers do not actively participate.



We use computers mainly because they are able to perform certain tasks incomparably faster and more accurately than people. However, computers cannot guess what kinds of tasks we have in mind or how to help us perform them. We have to tell them that. How? It's best to do so in a similar way as when passing information to other people, in other words, by using a language that is understandable to both sides. Using such a language, we write a program, a formalized solution to our problem, which can be executed by the computer.

Unfortunately, a language that is directly understandable by a computer will be absolutely illegible to a regular human being. It would be a sequence of bizarre instructions, written in numerical form, referring to computer components we didn’t even know existed (and frankly speaking we don't necessarily have to know). This way of communication, which dates from the beginning of computing, is nowadays used extremely rarely, and only in very specific situations.

So in order to help in communication with the computer, programming languages were invented that are somewhat similar to natural languages (that is, those used for communication between people). There are thousands of programming languages, and they differ in purpose (aside from general-purpose languages, there are many specialized ones, for example, languages designed only for statistical calculations), level of abstraction (in simple terms: the higher the language level, the less we need to know about the hardware on which the program is executed), ease of use, effectiveness of the programs written in them, etc.

#### 1.1.3 JavaScript as an interpreted language

JavaScript is a typical **interpreted language**. If we run a code written in JavaScript in a web browser, as it is happening, such as after loading the page we are currently reading (yes, yes, there are elements written in JavaScript on this page too), the interpreter will be the JavaScript engine built into the browser. This is not the only way to execute JavaScript code.

Perhaps you have heard of **node.js.** It is also an interpreter, but installed independently of browsers as an environment in the computer's operating system (it can be macOS, Windows, or Linux). Using node.js allows you to write programs in JavaScript that will, for example, turn your computer into a server.

At the beginning of this paragraph, we simplified things a bit. JavaScript is an interpreted language – there is no doubt about that. And in fact, running a program written in JavaScript looks as if we are executing our source code (that is, the code that we wrote) step by step. However, you may come across information about this language, and more specifically about particular interpreters, that is a bit different.

Most modern JavaScript engines use the *Just In Time Compilation* technique (*JIT Compilation*). This technique consists of compiling code fragments during the execution of the program (more than a single instruction) and allows you to increase its performance. However, from the user's point of view, such a change is virtually unnoticeable – it still looks as if only the interpreter is executing the source code, instruction by instruction.

Regardless of the language you choose, a few things remain the same while writing the program. First of all, an important, and probably the most difficult, stage of this process is to correctly define the problem we want to solve. Only then do we try to find the optimal solution, which we will finally present in the form of a program written in the chosen language.

So before you start explaining something to the computer, in other words, writing a program, you need to understand exactly what you want to achieve and how you want to achieve it. Secondly, the solution we propose and write in the form of a program must be 100% unambiguous – the computer cannot guess anything.

A simple example from a slightly different field: at some point in your life, you probably happened to buy a piece of furniture that required assembly. Assembling it is a problem that you, the buyer, have been burdened with. In order for you to cope with this task, you get a set of instructions to guide you through the whole process. You’re acting as an interpreter at this point, using a program that will allow you to complete the task. The success of your mission depends on the quality of these instructions, whether they are precise and unambiguous, and do not come from another piece of furniture. In the end, it may turn out that you have constructed not the furniture of your dreams, but a surrealistic construction from another dimension.

For the instructions to be good, someone who develops them must know exactly what they should illustrate, in what order certain actions should be carried out, at which stages something is easiest to confuse, and so on. And of course, they must know what effect is to be achieved at the end.

#### 1.1.4 A few more words on JavaScript

As we mentioned before, JavaScript is an interpreted programming language. Like most interpreted languages, it is also a high-level language (i.e. relatively easy to understand for people and separating us from the hardware details).

Back in the early 90s, all web pages were static. Things changed in 1995 when the Netscape corporation hired Brendan Eich, and tasked him to develop a new language for their product, the Netscape Navigator web browser. The new language was called LiveScript, but soon after its name was changed to JavaScript. Its main task was to add dynamics to websites, which would allow, for example, for more complex interaction with the user. And so the career of JavaScript began.

#### 1.1.5 Client-side vs server-side programming

The use of JavaScript on websites, which over time has become more and more complex and often contain very sophisticated logic, is called client-side programming. The code to be executed is loaded together with the page in the browser, on the user's side, and the interpreter which is a part of the web browser allows for its execution.

Today, JavaScript is the sole language supported by all major web browsers, and about 95% of web pages worldwide embed JavaScript code within them. From the beginning, web pages used JavaScript on the client-side to add interactivity and dynamically change the content.

Now it’s much more than that, as JavaScript offers many great frameworks on which to build huge, complex web applications and social networks (you've probably heard the names of frameworks like React or Angular).

All this can work on a variety of equipment, from high-performance workstations to simple smartphones. Thanks to the power of JavaScript, we can order food, play browser-based games, watch movies on streaming platforms, and be in constant contact with the people important to us. JavaScript is so popular that continually more and more effort goes into using it, not only as a client-side solution.

Over time, JavaScript began to appear in other areas, such as programming the server-side parts of complex web applications, also called back-end. These programs are executed on servers, processing data (e.g. from databases), which after processing will be available on the client side. The flexibility of this language and its relative simplicity have made it much more applicable, for example, in mobile apps, or even in programming UAVs (some drones run programs written in this language).

#### 1.1.6 Is this the perfect programming language? – disadvantages

We say that JavaScript is a mature language, which means that most of the features are already implemented and stable, and we will probably not see any big changes in the language. Since 2015, many aspects of JavaScript have changed, and many new features have been added. A lot of these changes were introduced to make the migration to JavaScript easier for programmers who know other popular languages, from which JavaScript originally differed quite strongly in certain aspects, such as when handling objects. We can still use the language in the old way, but it is recommended rather to use the modern JavaScript.

But... there are no ideal solutions, so there are no good programming languages for all applications. Each of them has its own limitations, and it’s no different with JavaScript. Despite its popularity and success, JavaScript is not a perfect programming language. Due to its nature, it is not suitable for certain applications. For example, there is no point in using it to write programs that require advanced mathematical calculations or very high performance.

Some limitations are due to the very concept of the language, but the vast majority are related to the platform on which we use it. This is especially visible when writing code to be executed in a browser, which as we said earlier is called client-side. In such a situation, JavaScript is limited in functionality by the fact that browsers, for security reasons, run script code in a sandbox environment (an environment separated from the outer world), which doesn’t allow for access to local files and resources (i.e. those files that are on the computer where the browser is launched).

Another inconvenience is that since the code is not compiled, it goes into the browser in the same, or a very similar, form to what we wrote ourselves. Why is this a disadvantage? This is because everyone can see our solution in an easy-to-read form and use it (either fragments of it or even the whole of it) without our permission to write their own program.

Some help here may be code obfuscation, which consists of transforming our ready script into a slightly less readable form (e.g. by generating short random names of variables and functions, eliminating end-of-line signs, and so on), but the simple fact is that if somebody wants to steal our JavaScript code, there is very little we can do to stop them.

#### 1.1.7 Is this the perfect programming language? – advantages

On the other hand, JavaScript has many advantages over other programming languages, and one of the biggest is a very active and supportive community. It is easy to find solutions to common problems, and to find help in general. This also means that tools that work with JavaScript are actively developed.

Another big plus is a huge number of ready-to-use frameworks and libraries that provide most of the commonly required functionalities and features. The language itself is relatively easy to learn, and allows us to focus on the job instead of fighting with the syntax (that is, the way of building the instructions which make up the code of our program).

Additionally, JavaScript doesn’t require you to buy expensive tools to work with it, and really good tools are already embedded inside your web browser. Last but not least, big players like Google, Facebook, and Mozilla actively support JavaScript tools and their development.

However, what is an advantage for some may turn out to be a disadvantage for others. An example may be the dynamic typing characteristic of JavaScript. In short, it consists of the fact that we can store data of any type in a variable (a variable is a container in which we store the data we will use).

For example, during the program's execution, we can store the number 10 in a variable, and in the next step use the same variable to store the "abc" string (deleting the previous value automatically, of course – don’t worry if you don’t understand right now, because we’ll be covering all these terms later on).

Usually this is very convenient, but a number of people have found this feature of the language to be a disadvantage. In their opinion, it makes it easier for a programmer to make mistakes in certain situations. By adding static typing, where a variable can only contain one type of variable (e.g. numbers) during program execution, a new language called TypeScript was introduced.

Remember also that if you learn to program in one language, it will usually be much easier for you to learn the next one, which for some reason may be better to solve a particular problem.

But let’s start now with JavaScript, which, due to its flexible and simple syntax, is perfect to learn as a first language.

#### 1.1.8 Let's get ready to work

As we mentioned earlier, JavaScript can be used in various environments, although most often it will be a web browser or a server with a node.js environment.

Each environment imposes a slightly different way of using this language, and some mechanisms or functions characteristic to it appear.

However, the essential part of the language, its core, remains the same.

In this part of the course, we will learn how to program using this invariable, core part of JavaScript: how to declare variables, write functions, conditional instructions or loops – all of this will be equally usable in any environment in which we decide to use this language.

Programming in any language is not an easy thing to learn, and you may feel overwhelmed by so much new information. If you’re persistent and focused, you’ll be writing simple scripts in no time, and there is no other way to learn to program than to write lots and lots of code.

Most importantly, don't give up even when you’re stuck: take a break, go for a walk, return to it with a fresh mind, and try again. In the end, slow and steady wins the race.

Now, let's begin!

### 1.2 Section 2 – Setting up programming environment

#### 1.2.1 Development tools

Like any other task, programming requires the proper tools and workspace. Software development, in most cases, requires a code editor and a compiler or interpreter of a given language. This is a minimum set, which we can extend as needed with various other tools.

At this stage of the course, apart from the JavaScript code editor and interpreter, we can also use the debugger, which is a tool that allows us, among other things, to pause the program in the indicated place and analyze its current state (e.g. the values of the indicated variables).

The tools in question will of course have to be run on the computer. At this stage, its performance is not particularly important, and any unit that can handle normal office tasks will suffice, so it's highly recommended to work from a desktop or laptop computer.

There is no denying that the size of the monitor will affect the comfort of your work. The bigger the monitor, the easier it will be to place the code editor, interpreter, and other content (e.g this course) next to each other. In normal working circumstances, programmers very often use several monitors.

The operating system doesn’t matter, as the appropriate tool can be found for Windows, macOS, and Linux.

At this moment, there are two choices. You can install all the necessary tools on your machine and work in the local environment. This is the preferred approach, as this is how it looks in real commercial projects most of the time. You can also customize everything to suit your needs.

Another approach is to use online tools. These can be convenient, as you don't need to install or configure anything – they just work. Most of them allow you to store your work in a cloud so you can access it from different devices, but on the other hand, they lack customization options, and you need to have a constant internet connection.

All the code that you’ll see in this course was tested in both local and online environments, so both options are valid. Finally, we can move on to the choice of tools.

#### 1.2.2 Online development environment

**Online environments**, commonly known as code playgrounds, are sites that act as a simple editor and runtime environment. All of them have similar sets of features. They have different user interfaces, but in principle, they behave in a similar way. They allow you to write code, run it for testing purposes and most often share it with other users.

In the case of JavaScript, where preparing a fully working local environment actually boils down to installing a code editor and running the browser, they are not as important as regular development environments. They are mainly used as training and testing platforms, or places to publish sample solutions to programming problems.

Among JavaScript programmers, the most popular are the following:

* [*JSFiddle*](https://jsfiddle.net/)
* [*CodePen*](https://codepen.io/pen/)
* [*JsBin*](https://jsbin.com/)
* [*Plunker*](https://plnkr.co/)

During the course, we will use an online environment integrated with the training platform. OpenEDG provides a simple environment for writing and running code in several programming languages, including JavaScript. Thanks to that, you will be able to practice everything we talk about right away.

Don't forget, however, that this platform is a purely didactic and testing solution, and it certainly cannot be used as a fully-fledged development environment. However, it is ideal for our needs, because in most cases we will be able to forget about the web surroundings of programs written in JavaScript, including HTML elements. This will allow us to focus solely on learning the JavaScript language itself.

However, it is highly recommended that you also set up your own **local development environment**. It's not difficult, as you'll find out right away, and it will allow you to do some exercises in a way that is much closer to how you would do it during normal software development. If, in the further part of the course, any of the exercises will have to be done in such an environment, we will indicate this clearly.

#### 1.2.3 Local development environment

As we wrote earlier, the JavaScript requirements for the development environment are very modest. In most cases, especially at the beginning of development, just three elements are sufficient: a code editor, an interpreter (i.e. a bootable environment) and a debugger.

Depending on the level of sophistication, the complexity of the written project, or the environment for which we write our programs (client-side, server-side, mobile), other tools may also be needed.

These will be, among others:

* **package managers** – enabling the management of libraries (containing ready-made solutions that we can use in our programs) or components of the development environment (e.g. npm or yarn)
* **task runners and module bundlers**– used, in simple terms, to automate the process of software development and merge the resulting code from many files and libraries (e.g. Grunt or Webpack)
* **testing framework** – allows for automatic testing of the correctness of our program in search of potential errors (e.g. Mocha, Jasmine, or Jest)
* **security analyzers** – as you can guess, used to control the security of our solution (e.g. Snyk, RetireJS, or OWASP Dependency Check)

The openness of web development environments is both a blessing and a curse. We have a choice of hundreds of components, from which we can create the most comfortable environment for ourselves.

However, their quantity, plus the dynamic changes of particular tools or even just the trends among programmers make it difficult to keep up with everything that’s happening within these environments.

But for us, this is a problem for the distant future.

For now, we need the minimal trio: a **code editor**, **interpreter**, and **debugger.**

#### 1.2.4 Code editor

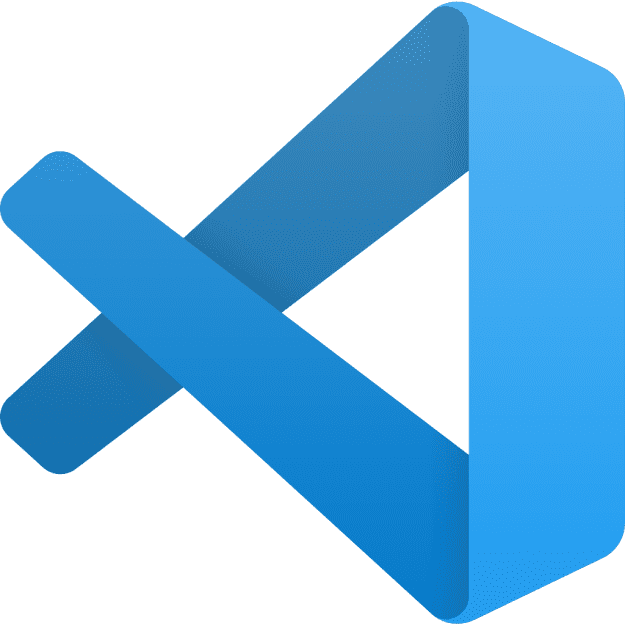
The code of almost all programming languages is composed of some form of text. So, to write the code, we need a text editor. But it needs to be an application that writes plain text (it can’t be a rich text editor, like MS Word). In other words, just a plain notepad that can write .txt files is enough to write code, although it’s much easier if you use a dedicated code editor. The market is full of professional code editors, both free and paid. Some of them are universal, while others are exclusive to specific languages. The main advantage of using a dedicated code editor is syntax highlighting, text autocomplete, and error checking. This improves work efficiency and code understanding, and lowers the number of errors and typos. There are many good code editors, but it can be really hard to select one that works for you.

Here are some popular ones:

* [***Visual Studio Code***](https://code.visualstudio.com/)

[Windows, macOS, Linux]

Powerful code editor free for both personal and commercial use. It has quickly become one of the favorites when it comes to web development. It has built-in features like a JavaScript debugger, and tools to streamline web projects. It’s also highly customizable via the extension system (there are many additions dedicated especially to the JavaScript language).



* [***WebStorm***](https://www.jetbrains.com/webstorm/)

[Windows, macOS, Linux]

A popular commercial development environment, in which the code editor is just one of the smaller elements in a huge set of tools that improve code development (e.g. supporting testing). Intended for large projects, it may prove to be too heavy and complex for small programs. Although it is intended for commercial use, it is possible to obtain a free educational license.



* [***Sublime Text***](https://www.sublimetext.com/)

[Windows, macOS, Linux]

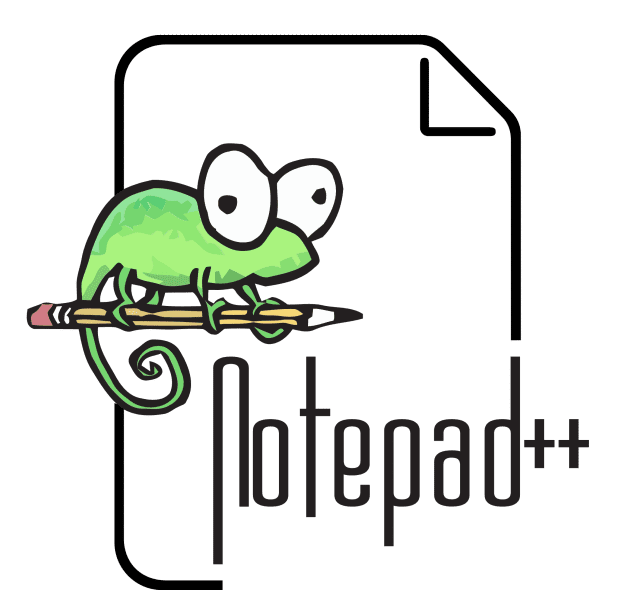
Fast and easy-to-use code editor with many advanced features, like multiple-line editing, fast search, and others. A trial version is available, but for long-term usage, a license needs to be purchased for both private and commercial use.



* [***Notepad++***](https://notepad-plus-plus.org/)

[Windows]

Free and lightweight code and text editor. The program is small and fast, supports dozens of programming languages, and can be extended with plugins. It may be old and ugly, but it’s still sharp.



Lots of other code editors exist, both free and paid, and you can use whichever one you prefer. Many developers use, among other things, console editors, including the legendary vim. Console editors are not run in a graphical environment, but in a text console. However, you can only reach for such solutions if the tasks you're going to do turn out to be too simple and you want to make your life a little bit more difficult.

#### 1.2.5 Interpreter

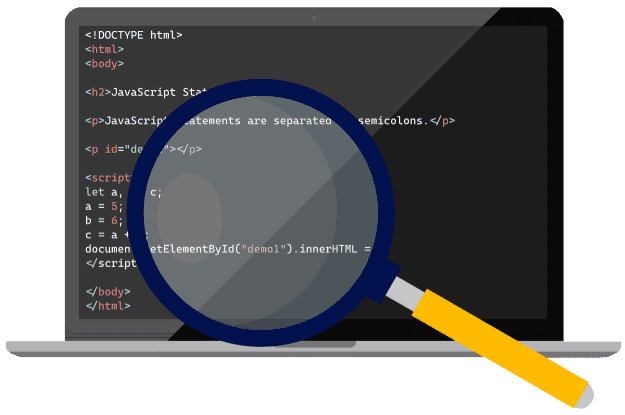
We have already talked a bit about the **interpreter** and its role. It functions as a runtime environment for our program. It checks whether we have made any formal errors, for example, making a typo in the name of a function or forgetting to close a parenthesis, and then it executes the program instruction by instruction.

The choice of JavaScript interpreter will depend on which platform we write our software for. For example, if we want to write a simple server-side application, we will almost certainly choose the **node.js** environment, which we will have to install directly on our operating system. In the case of client-side software, our interpreter will simply be the web browser you already have installed (because how else would you read this course?).

Our course is about core JavaScript, that is, those language elements that will be equally useful in client-side, server-side, and mobile solutions. So we can practice them in any environment, using any interpreter. The easiest way to do this is to limit yourself to a web browser.

As we said before, practically all browsers have built-in JavaScript engines (or interpreters), but we strongly recommend using **Chrome** from Google, or **FireFox** from Mozilla. Both are known for their efficiency and integrated advanced tools for web developers (that’s you). They are available for Windows, macOS, and Linux.

Remember to regularly update your chosen browser and use the latest version. This is especially important when working with JavaScript. The language is constantly changing, with new features and mechanisms being added. You may find that your favorite but somewhat dated browser doesn't support certain features of the language. Your browser is now a tool, so try to keep it in good shape by updating it regularly.



#### 1.2.6 Debugger

Computer programs are complicated beasts, thousands or even millions of lines of code (but calm down, we'll start with just a few). With such complexity and size, it’s impossible to produce code without any errors. Some types of errors, especially logical ones (formally, the program is written correctly, but probably we invented the wrong solution to the problem), can only be found while the program is running, and often only in special circumstances. It’s really hard to find out what exactly is happening inside a program that runs blazing fast, and for those problems, debuggers exist.

A **debugger** is a tool that allows you to slow down or even halt the execution of a program, run instructions step by step, and look at and analyze the state of the program at any given moment.

Fortunately, the moment we decided to use the web browser as our boot environment and JavaScript interpreter, we also got ourselves a debugger. All modern browsers are equipped with the developer tools. During normal operation, they are invisible, and we have to enable them in the browser options (more about this in the next chapter).

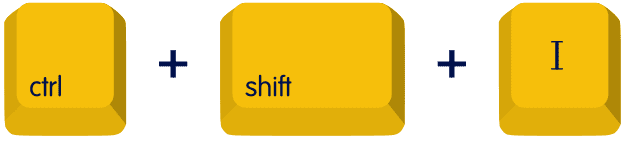
Depending on the browser, we will find various tools there, but there will certainly be:

* **the inspector** – which will allow us, for example, to analyze the individual HTML elements of an open website;
* **the JavaScript console** – which firstly shows all the information about the errors, and secondly allows us to run single JavaScript commands in the context of the current page;
* **the debugger** – which, among other things, shows the current values of variables, and allows you to pause code execution in the indicated place and to perform step-by-step work (i.e. execute single instructions of the program).

How do you enable the developer tools? Unfortunately, there is no single answer; it depends on the browser you’re using (sometimes also on its version) and the operating system. Browser interfaces change quite often, so it is better to learn the right shortcuts instead of looking for the right option in the menu.

Try the following key combinations:

* Windows and Linux operating systems, all common browsers except Internet Explorer and Edge:



* Windows operating system, Internet Explorer and Edge:



* macOS operating system, all common browsers:



In the next chapter, we will return to this topic and learn a few more things about these useful tools.

In the next chapter we will write our first piece of JavaScript code. We will test it first of all in the runtime environment integrated with our training platform. We will also use it to check how our local development environment works. So, please make sure that your selected tools are installed and that you can start them up. If you still don't know what to choose, we suggest using the local environment with **Visual Studio Code** (our code editor) and the **Chrome** (web browser with JavaScript interpreter and debugger).

### 1.3 Section 3 – Hello, World!

#### 1.3.1 The "Hello, World!" Program

Why "Hello, World!"? For almost 50 years, this sentence, and its derivatives, has marked someone as learning a new programming language, although it’s more a tradition than anything else. The phrase was used a long time ago in a very important book about the C language, but the text itself doesn’t matter.

The idea is to write out something on the screen using a specific language. First, it allows us to see the basic syntax of the language and compare it to other programming languages. Second, it’s a very simple program, and anyone can easily write it or copy it from the internet and check if their tools and environment are set up correctly. Third, it’s a program that outputs something, so it provides feedback on whether it was executed correctly or not.

In the case of client-side JavaScript, displaying something on the screen can be understood in two ways.

First, client-side JavaScript is always executed in the context of a website and allows you to manipulate elements of that website. So we can, for example, use the appropriate function to insert some text, change a title, create a table, etc. on the page. This way, we control the visual part of the website.

Second, we can use the **console** as a screen to write some information. The console, as we mentioned in the previous chapter, is part of the developer tools. So it is not visible by default, and it must be properly enabled (we also wrote about this in the previous chapter). For our needs, it will be much more convenient to use the console, as we will avoid the need for a thorough analysis of the structure of the website.

But what actually is a console? First of all, it is a place where various messages are displayed, normally invisible to the browser user. These messages can, for example, be generated by the JavaScript interpreter after encountering an error or if we print it, by calling the appropriate function. Secondly, we can run individual JavaScript commands in the console, which will be executed in the context of the currently loaded web page (a little more about that in a moment).

The basic function allowing us to write information to the console is **console.log.**

So, to refer to the eternal **"Hello, World!"**, we should call it as follows:

console.log("Hello, World!");

We can treat the **console.log** as a function\*. In fact, the function is only a log, and console is the object to which the function belongs.

\*This type of function, belonging to an object, is generally called a method. But once again, for the time being, to simplify certain things, let’s assume that this is an ordinary function – it will not bother us at all (we’ll learn about objects much later).

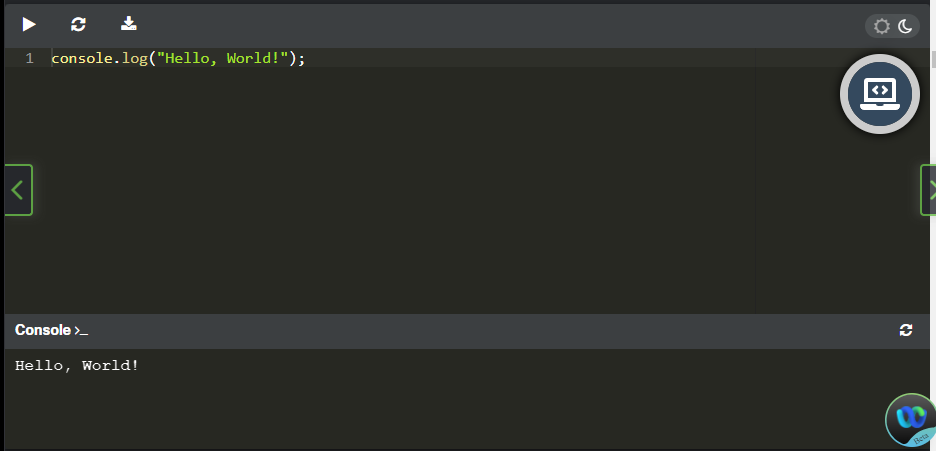
The **function** is a piece of code that allows you to perform a specific task (in our case, to display something on the console). Functions often take arguments, in other words, data that they will use during operation. In JavaScript, we run a function by calling it, and we call it by writing its name followed by a pair of parentheses, where the arguments are provided (if the function does not need arguments, the parentheses are left empty). In our example, the argument is the text we want to display. Note that in order to indicate that "Hello, World!" is the text, we put it in quotation marks.

To let the interpreter know where the command ends, we put a semicolon at the end of the function call. In this case, the interpreter would manage without such help, but it is a good habit to end each command with a semicolon, so that you do not forget it when it is really needed.

We already know what to write, and the only question is, where to do it?

#### 1.3.2 Online development environment

In the editor, you should see the piece of code that was just discussed, containing the **console.log** function. Try to run it. You need to press the highlighted button with the play icon, located directly above the editor.



As a result, the lower window simulating the console should show:

Hello, World!

**Output**

Go to the editor again and change the word "World" to your name. Start the program again and check what appears in the console window. Congratulations, you just modified a program written in JavaScript.

We could discuss your first JavaScript program based on this example. You've learned its syntax, run it online, checked its effect, and even modified it yourself. You can test all the examples that we discuss in this course in this way. However, in the chapter before, we urged you to configure your local development environment. So it would be good to show you how this example can be run in such an environment. And this will require a slightly longer introduction.

#### 1.3.3 Local development environment

Client-side JavaScript is a language of the web and exists only in the web ecosystem. In this setup, JavaScript cannot exist by itself. JavaScript code needs to be embedded in an HTML document. When we used the online environment to run our program, certain aspects were hidden from us. This time we will have to look at them more closely.

**A few words about HTML**

**HyperText Markup Language**, or **HTML** for short, is a set of tags used to describe the structure of a website. It allows us to give a page the format of a document containing sections, headers, paragraphs, lists, and the like. HTML is definitely beyond the scope of the current course, so we will present only some basic information about it, just enough for you to understand where and how we can run the JavaScript code associated with a given page.

Tag types are predefined. For example, the tag that specifies a paragraph is <p> and the tag for the first degree (largest) header is <h1>. The tag name must be placed in angle brackets. Tags are usually used in pairs, limiting a certain area of the document (we have an opening and a closing tag). The closing tag is different from the opening tag, because a slash appears before the name. For example, a paragraph may look like this:

<p>It's just an ordinary paragraph.</p>

Often, tags can (and sometimes must) be placed within the range of other tags. For example, our paragraph should be placed inside the <body> tags, which separate the main part of our document.

<body>

<p>It's just an ordinary paragraph.</p>

</body>

**Minimal HTML document**

Let's try to create a minimal HTML that defines an empty page.

<!DOCTYPE html>

<html>

     <head>

     <title>Empty Page</title>

     </head>

     <body>

     </body>

</html>

Let's start with the declaration **<!DOCTYPE html>**. This is not a typical tag, as it is used to inform the browser that the whole document has been prepared according to HTML5. The actual document description starts with the **<html>** tag, which together with the **</html>** tag sets the boundaries of the document. Every other tag should be inside these. If any given tag holds other content, there will be a corresponding closing tag, forming some sort of container.

The next tag, **<head>**, contains additional information about the document, which must also be placed in tags. The most basic one is the **<title>** tag, which sets the title of the page mostly visible in the browser title bar. After **<head>** there is the **<body>** element, and the visible content of the web page should be placed there (e.g. our paragraph).

#### 1.3.4 The script tag

The JavaScript code to be executed by the browser on the page must be attached to the HTML using the **<script>**tag, and there are two ways it can be done. The code can be embedded directly inside of the **<script>** and **</script>** tags, but this is only recommended when the code is short. Another approach is to use the **"src"** attribute to point to a separate file that contains the JavaScript code. This is especially true when the same code is going to be used on several pages, because repeating exactly the same code many times is bad practice, as any changes need to be applied to all the files; and additionally, it artificially increases the page size. The JavaScript file extension is .js.

HTML is read by the browser line by line, and script tags are executed right at the moment when the browser parses the **<script>** tag (parsing for programming languages means a formal analysis of the code by a machine in order to understand its structure). Usually **<script>** tags are inserted in the page header between the **<head>** and **</head>** tags, and we can insert many of them in a file, for example, in order to include JavaScript code from different files. This behavior can be changed for external scripts pointed to by the **"src"** attribute by using the **"defer"** or **"async"** attributes.

* **defer** – means that the script should be executed after the whole page is loaded;
* **async**– means that the script will be executed immediately, but in parallel to parsing the rest of the page.

#### 1.3.5 ... and a little something about CSS

**CSS**, or **Cascading Style Sheets**, is a language used together with HTML to describe the appearance of a page and its elements. In a nutshell, HTML describes the structure of a document, while CSS describes its presentation.

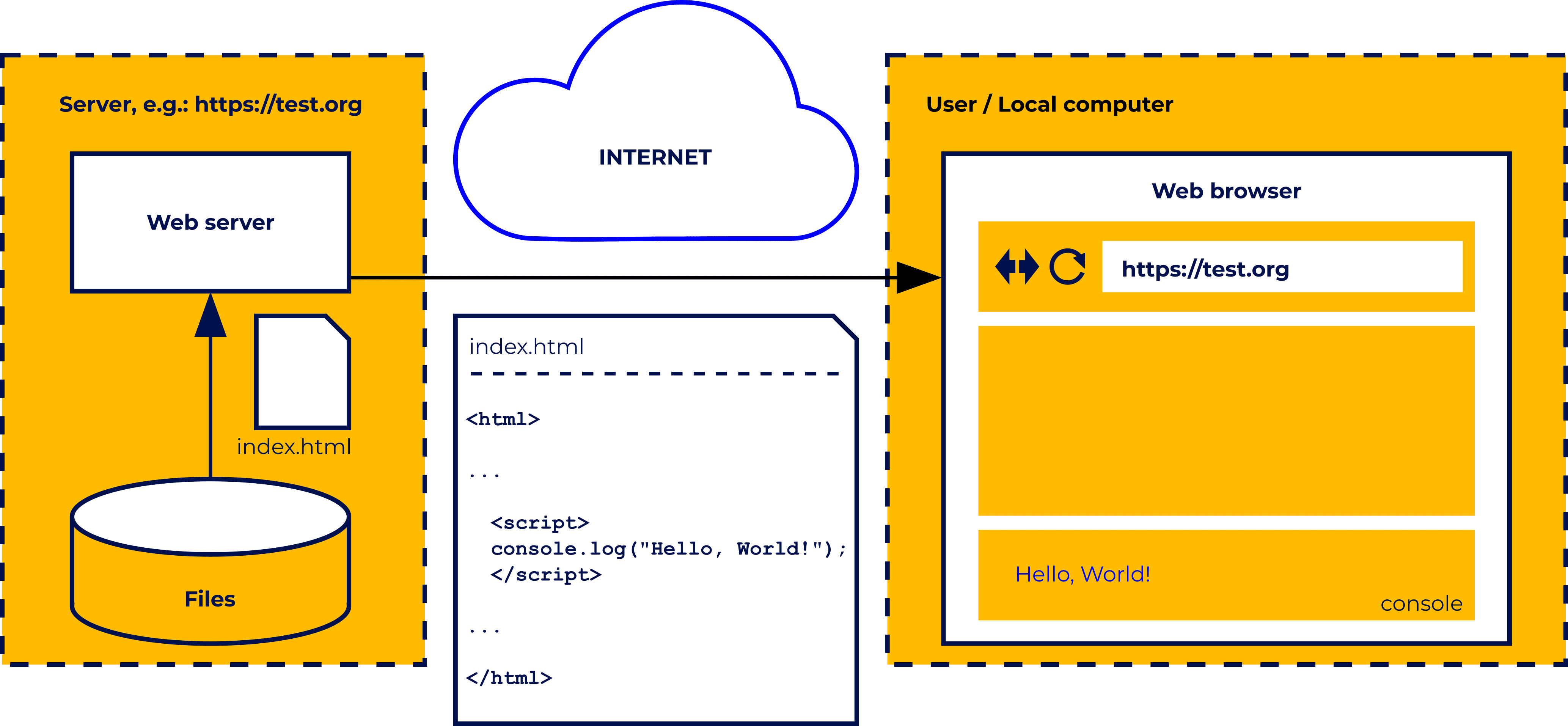
For example, in HTML, we can describe a page that has a header, two paragraphs, and a data table.

In CSS, we can define what font will be used on the whole page, what color the background will have, or whether the mouse cursor, when moved over the table, should change shape.

So we can treat CSS as some kind of configuration of the visual layer of the page. Thus, the website will most often be built on the basis of an HTML file (that is, a description of the structure), JavaScript code (allowing us to add, for example, some interaction mechanisms) and a CSS file (describing the presentation layer of the page). However, what is important is that there will be no page without an HTML file, but we can easily create a page without using CSS files. The CSS description itself is outside the scope of the current course, and we mention it only for the sake of order.

#### 1.3.6 How can we run our JavaScript code?

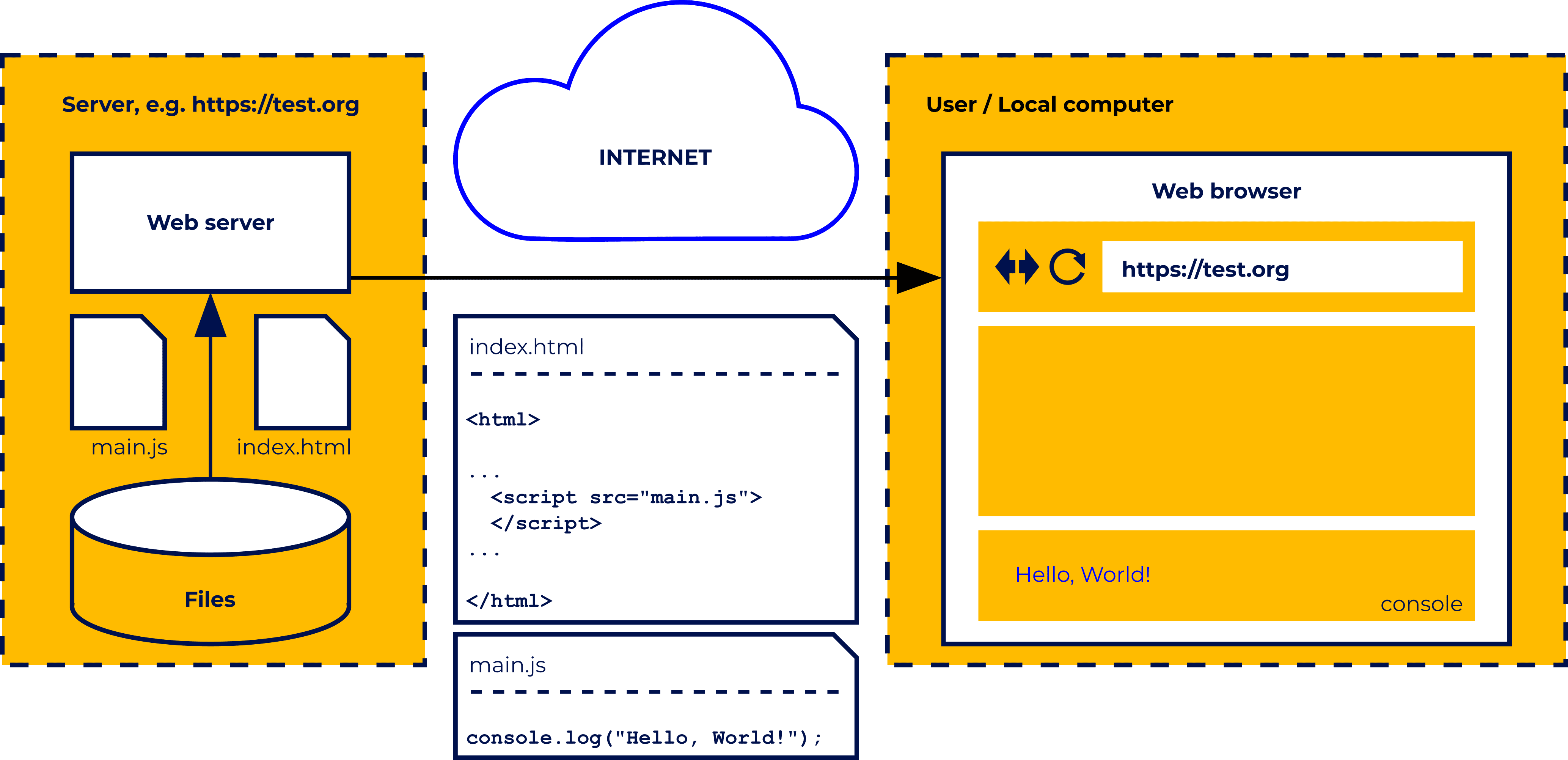
Let's start with a simple example, where the browser gets a simple (maybe even an empty) page from *https://test.org*. The address is made up for this example, so don't really try to enter it. Look at the figure below.



Let's start from the right side of the figure. The user runs a web browser on their computer (e.g. Chrome). Using the appropriate hotkey shortcut, they turn on the **developer tools** (see the previous chapter) to be able to use the console. Remember that these tools are not needed for normal browser use, and are therefore hidden by default. Then the user types *https://test.org* (the URL of our fake site) in the address bar.

On the remote server (left side of the drawing), associated with the address *https://test.org,* a web server is launched, which, after receiving a request from our user, will prepare an answer for them. In the simplest case, the answer will only contain an html file, which can be stored on the same server. The html file (in this example, *index.html*) is sent back to the user and processed by the browser. If some content (e.g. a paragraph with text) is defined in it, it will be displayed in the browser window.

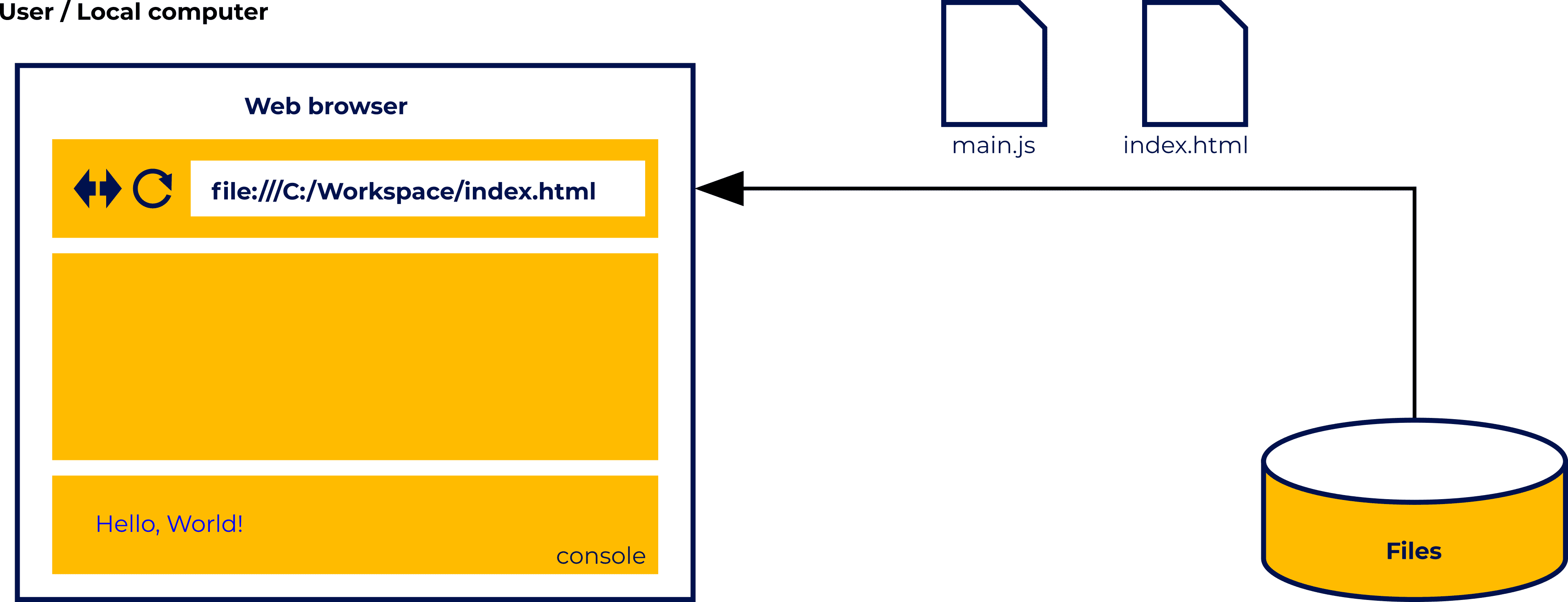
However, we are more interested in the fact that the index.html file contains the **<script>** and **</script>** tags, with a piece of JavaScript code between them. Do you recognize it? This is obviously an attempt to show our "Hello, World!" on the console. When loading the page, the code placed inside the **<script>** tags should be executed and, if the developer tools are enabled and the console panel is visible, the console will show "**Hello, World!".**



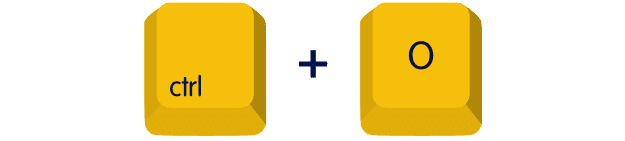
Everything will work exactly the same as in the previous scenario, except that the web server will provide the main.js file in addition to index.html. The user will not notice any difference. Of course, placing our code on a remote server just to test it would be a bit cumbersome.

We have another possibility, which is that we can load a local html file (i.e. one that is on our computer) into the browser. If this code contains a **<script>** tag indicating some JavaScript file, then this file will also be loaded from the local resources.

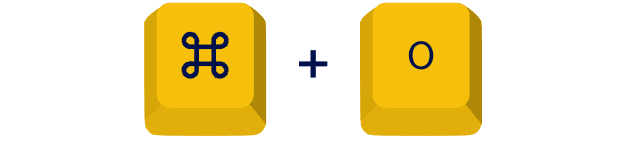
The picture shows a simple scenario in which the user loads a local index.html file into the browser, in which there is a reference to main.js (so this file will automatically be loaded, too).



You can load a local html file either by typing its local path after file:/// in the address bar, or by simply opening it in your browser using the Open command from the menu. Since the menu in browsers is very often hidden, a simpler way may be to use a shortcut to open existing documents in applications. The shortcut is universal, not only for browsers and you have probably already seen it:



or in the case of macOS:



#### 1.3.7 Okay, maybe we could finally run something...

To run this locally, you’ll need to open the code editor of your choice. Create a new file with the extension .html (the name of the file doesn't matter, but it’s good practice to avoid spaces in the file name). Put the following code in this file and save it.

<!DOCTYPE html>

<html>

     <head>

     <title>Empty Page</title>

     <script src="main.js"></script>

     </head>

     <body>

     </body>

</html>

Then, in the same editor, create another file, this time called *main.js* (this is the name we used in our html file). It should contain one line you've seen before: we used in our html file). It should contain one line you've seen before:

console.log("Hello, World!");

Save the changes and go to the browser. Open a new tab, enable the developer tools (they open for a particular tab), and select the console tool. Give yourself a moment to get used to the layout of the developer tools (each tool, including the console, should be placed in a separate panel, which can be selected).

Just in case, make sure that the focus is set to the browser window (i.e. the new tab) by clicking on it. Then, using the appropriate keyboard shortcut open the html file you have just created. If everything has been done correctly, you should see our "Hello, World!" sign in the console. Make sure that it works and that you can see a message displayed in the debug console. If necessary, do it again, carefully following the steps.

And now a small challenge. Try to modify the html file yourself so that it does not refer to the main.js file. Instead, the same JavaScript code that we wrote in main.js should be placed directly after the <script> tag. If you have problems, go back to the first drawing in this section.

To run this code in the online environment, place it inside the HTML tabs, and if necessary, press the run button.

#### 1.3.8 Executing the code directly in the console

We have another, quite convenient option when it comes to running short pieces of JavaScript code in the browser (and our program, consisting of one instruction, is definitely short). As we said before, the console is not only used to display information, but also allows you to run individual JavaScript instructions. These instructions must be executed in the context of some HTML page. However, you don't necessarily have to write your minimal page, as we did a moment ago. Try to open a new tab and type **about:blank** in the address bar. This is a pseudo address that tells your browser to generate and load a minimal blank HTML page.

Then run the developer tools. At the beginning, we can check what the HTML generated by the browser looks like. To do so, select the first tool from the panel (in Chrome, it will be Elements, in Firefox Inspector). You should see absolutely minimal html code:

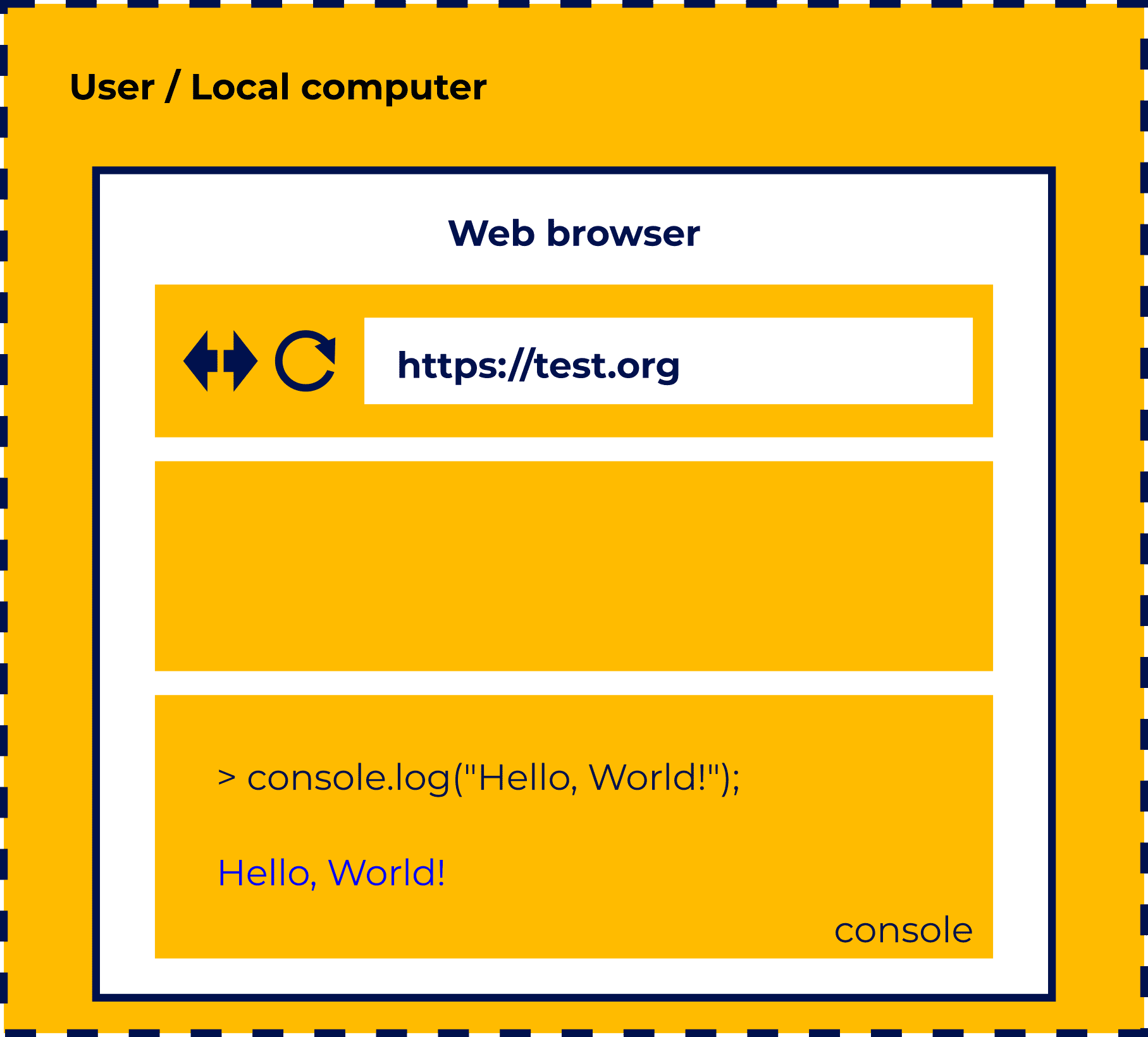
<html>

     <head></head>

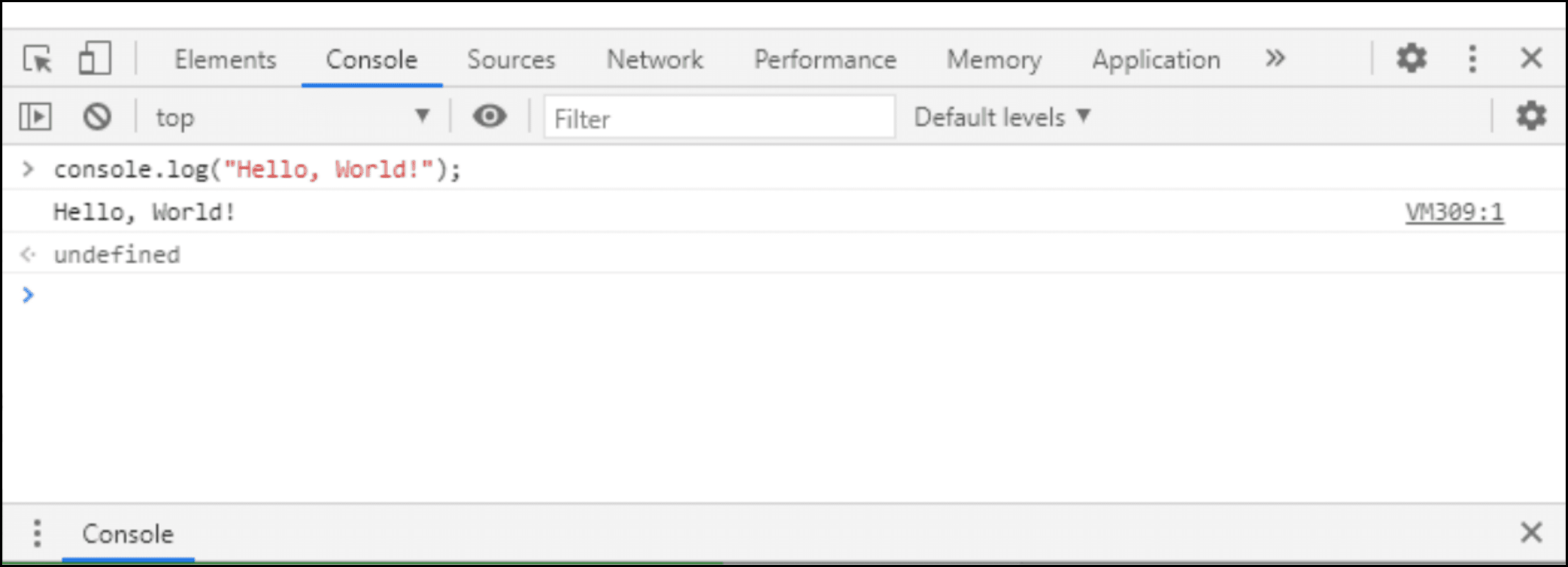
     <body></body>

</html>

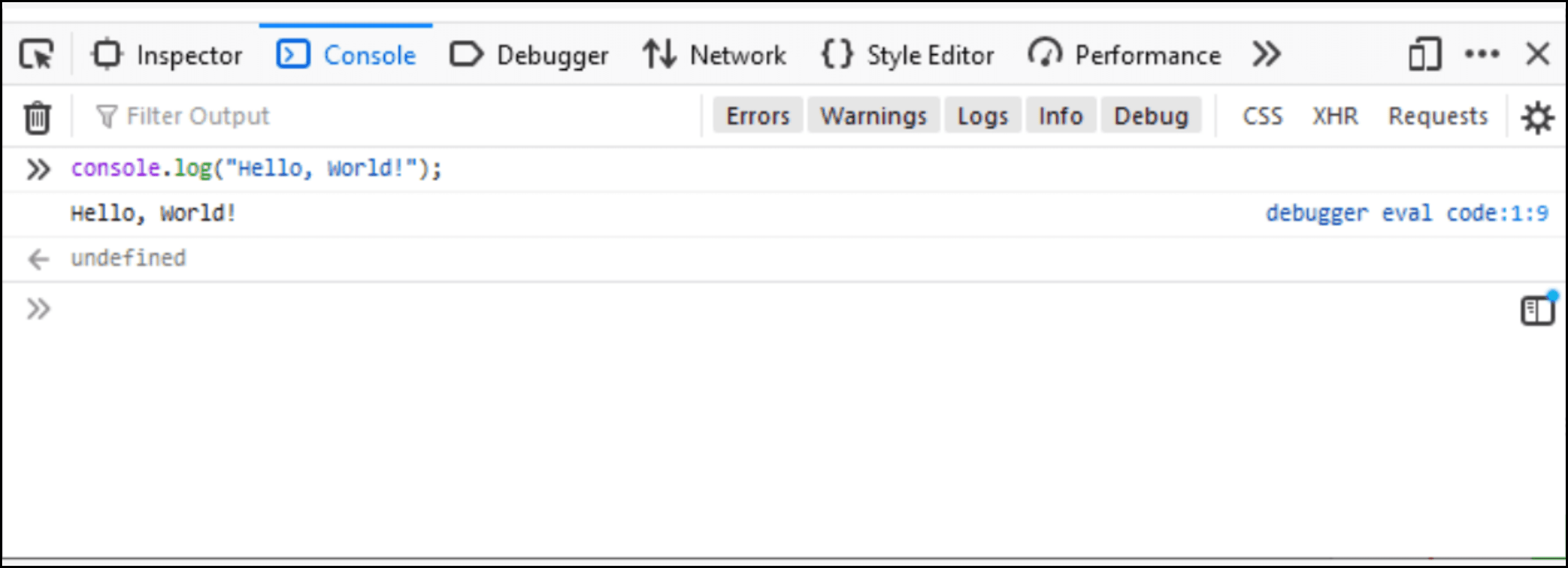
Now choose the console from the developer tools. You should see a prompt, usually a sign **>** or **>>** followed by a flashing cursor (if there is no cursor, click on the prompt). Then you can enter the instruction that will show "Hello, World!" on the console (using the console.log function). The scenario is shown in the figure below.



In fact, regardless of the browser, we should get the same effect – the console will display the text we specified. In the case of Chrome (running on the Windows operating system), the console should look something like this after completing this task:



In the case of Firefox (also Windows) this way:



For both browsers, the debugger windows containing the console may vary minimally depending on the version of the software and the operating system running it. The developer tools can be moved. They can be located at the bottom of the browser, as in the examples shown, but they can also be placed on the left or right side of the window (or as a separate window entirely). So don't be surprised if your browser's layout is slightly different from the pictures.

#### 1.3.9 SECTION SUMMARY

Our first program was launched in an online environment at the beginning. This environment allows us to hide certain details that are not important to us at this stage of the course. All exercises and examples that we will discuss should be done in this environment.

However, from time to time, it would be good for you to try to do the chosen example in the local environment as well. It is much closer to what is actually used in the work of a web developer. Running JavaScript code in the local environment may seem a bit cumbersome at first, but fortunately this is just a first impression. Remember, to test simple instructions, you only need to use the console with an empty page (e.g. about:blank). If you want to test a slightly larger piece of code, it is best to create a minimal html file that will refer to the file containing our JavaScript code using the **<script>** tag.

#### 1.3.10 SECTION QUIZ

**Question 1:** Use **console.log** to output your full name to the console.

Check

**Question 2:** Output your year of birth.

Check

**Question 3:** Try again to output your year of birth, this time passing the date without the quotes.

Check

**Question 4:** We can pass several arguments to **console.log** separated by commas, e.g.:

console.log("abc", "def", "ghi");

Output information about yourself to the console in the format: **Name Surname (Year)** e.g. *Mary Stuart (1542)*.

* giving all the information as one argument;
* giving the name, surname, and year as separate arguments.

Check

**Question 5:** Output the same information (name, surname, year) to the console, not side by side, but on consecutive lines.

Check

**Question 6:** A string can be concatenated using the**+** sign, for example **"abc"** **+** **"def"** will be treated as **"abcdef".** Try writing your name, surname, and year of birth on one line again, this time not separated by commas, but by **+** signs.

Check

**Question 7:** Put spaces in the appropriate places, so that when displayed, you get the same effect as in **Question 4**.

Check

### 1.4 Module 1 Completion – Module Test

The following test is based on what you have just learned. There are fourteen questions in total and you need to score **at least 70%** to pass.

Good luck!

**Question 1**

Multiple choice question

Where will we definitely not be able to execute JavaScript code?



In a server environment using *node.js*.



On a mobile device.



Directly in the processor.



In a web browser.

**Completed Question 2**

**Question 2**

Multiple choice question

The TypeScript language is:



an alternative name for JavaScript.



a variant of JavaScript that is compiled rather than interpreted.



a new language based on JavaScript which, among other things, introduces static typing.



the original name for JavaScript, which has been changed over time.

**Completed Question 3**

**Question 3**

Multiple choice question

The basic toolkit needed to effectively develop JavaScript code consists of two elements:



interpreter, package manager.



code editor, debugger.



code editor, interpreter.



interpreter, debugger.

**Completed Question 4**

**Question 4**

Multiple choice question

What is not the task of the interpreter?



Verifying the correctness of the program source code (syntax).



Transforming the individual commands of the source code into the target form.



Transforming all program code into target code before execution.



Creating a runtime environment for the program.

**Completed Question 5**

**Question 5**

Multiple choice question

Using JavaScript, we want to display the word **"test"** in the console. What statement do we use to do this?



**console (log, "test");**



**console("test");**



**log("test");**



**console.log("test");**

**Completed Question 6**

**Question 6**

Multiple choice question

A client-side JavaScript program:



requires server support to run.



may be embedded inside an HTML document or run standalone in the browser.



runs directly in a web browser and can contain an HTML document inside it.



should be embedded inside an HTML document.

**Completed Question 7**

**Question 7**

Multiple choice question

HTML is:



a network protocol.



one of the data-compression formats.



a scripting language to support mathematical calculations.



a language for describing the structure of a web page.

**Completed Question 8**

**Question 8**

Multiple choice question

The **<html>** tag indicates the start of the actual HTML document. What tag should appear at the end of the document?



**</html>**



**<html/>**



**<lmth>**



**<html>**

**Completed Question 9**

**Question 9**

Multiple choice question

What tag do we use in HTML to denote the main part of the document inside which we define the page elements?



**<head>**



**<content>**



**<main>**



**<body>**

**Completed Question 10**

**Question 10**

Multiple choice question

What HTML tag is used to indicate that we are embedding JavaScript code?



**<execute>**



**<script>**



**<source>**



**<js>**

**Completed Question 11**

**Question 11**

Multiple choice question

In a browser, we want to open a local file by typing the path to it in the address bar. The path must be preceded by:



**https://**



**local:///**



**http://**



**file:///**

**Completed Question 12**

**Question 12**

Multiple choice question

In the browser, we type into the address bar a string starting with **file:///**. This means that:



the rest of the string is the path to the file on our local computer that we want to open in the browser.



we want to save the page as a local file.



we want the browser to create an empty page for us with the name we give as a continuation of the string.



we do not want to open the page, but only download a file from the remote address and save it on the local machine.

**Completed Question 13**

**Question 13**

Multiple choice question

Entering **about:blank** in the address bar of your browser will:



generate a page with information about the browser's status and send it to the developer.



open a tab with information about your browser.



generate and load a minimal blank HTML page into the current tab.



reset the browser to its default settings.

**Completed Question 14**

**Question 14**

Multiple choice question

The JavaScript code includes the **console.log("http://test.org");** command. Its execution will:



display the following message on the console: **"http://test.org"**.



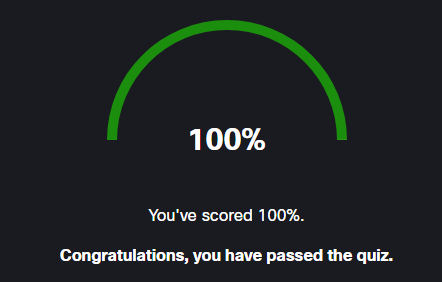
cause the page *http://test.org* to be loaded into the browser.



send a log with information about the currently executed script to the indicated address *http://test.org*.



display on the console information about the progress of *http://test.org* page loading.



## JSE: Module 2: Variables, Data Types, Type Casting, and Comments

### 2.0 Section 1 – Variables

#### 2.0.1 Variables

The ability to write various information on the screen, such as "Hello, World!" can be fun for a while, but it is not a universal way of writing programs. It's time to start learning more about the puzzle elements that will ultimately allow you to create programs that solve real problems.

There are quite a few of these elements, and we’ll introduce them gradually, although not necessarily in a simple chronology. We will often come back to what has already been discussed, extending the previous information with something new. Sometimes we will also go forward, using mechanisms that will only be explained in more detail over time.

At first it may seem a bit overwhelming, but with time everything should start to merge into one coherent picture.



The first element of programming that we will talk about is the **variable**. You may know the name of a variable from mathematics, where it means a symbol used as a placeholder for different values that can change. They have a similar role in programming.

What do we actually need them for? As you can guess, most programs are quite complex, and we are rarely able to solve the problem with a single operation. Usually, the program will consist of many more operations, each of which can produce some intermediate results, which will be needed in the next steps. Variables allow us to store such results, to modify them, or to feed them into subsequent operations.

#### 2.0.2 Naming the variables

Imagine variables as containers in which you can store certain information (such information will be called variable values). Each container must have its own name, by which we will be able to clearly indicate it.

Usually, we have quite a lot of freedom when it comes to inventing these names, but remember that they should refer to what we will store in the variable (e.g. height, color, stepCounter, and so on). Of course, JavaScript will not verify the correlation between the name and the contents of the variable – it is simply one of the many good practices that make it easier for us and others to understand the code later on.

In most programming languages, a variable must be declared before use, and JavaScript is no exception. Declaring a variable is simply "reserving" the name of the variable. This way, we inform the program that in the further part of the execution, we will use this name to refer to our container, in order to retrieve a value from it, or save a value to it.

In JavaScript, variable names may consist of any sequence of letters (lower-case and upper-case), digits, underscore characters, and dollar signs, but they must not start with a digit. There is a list of reserved words that cannot be used as variable names (look at the table below).

The important thing is also that the JavaScript interpreter distinguishes between lower-case and upper-case letters, also in variable names, so names such as **test**, **Test**, or **TEST** will be treated as different.

|  |  |  |  |
| --- | --- | --- | --- |
| The names of variables in JavaScript can be virtually any character string. However, there is a set of reserved words that cannot be used to name variables, functions, or anything else. They are integral parts of the language and are assigned meaning that cannot be changed. Below you will find a list of them. | | | |
| abstract | arguments | await | boolean |
| break | byte | case | catch |
| char | class | const | continue |
| debugger | default | delete | do |
| double | else | enum | eval |
| export | extends | false | final |
| finally | float | for | function |
| goto | implements | if | import |
| in | instanceof | int | interface |
| let | long | native | new |
| null | package | private | protected |
| public | return | short | static |
| super | switch | synchronized | this |
| throw | throws | transient | true |
| try | typeof | var | void |
| volatile | while | with | yield |

#### 2.0.3 Declaring variables

As we mentioned before, we **declare** the variable to reserve a name for it. This is a simplification, because in fact, memory space is also reserved for the variable, but when programming in JavaScript, we practically never have to think about what happens in the memory. Usually, the values stored in the variable will be able to be modified during the execution of the program (they are "variables", after all). Why usually? Because we can declare variables whose values cannot be changed. To be honest, we don't even call them variables anymore – we call them **constants**. For the declarations, we use the **var** or **let**for **variables** and **const** for **constants**. For now, however, let's stay with the usual variables, and we will return to the constants in a moment.

Let’s analyze the following code sample (you will also find it in the editor window – run it there and watch the results in the console):

var height;

console.log(height); *// -> undefined*

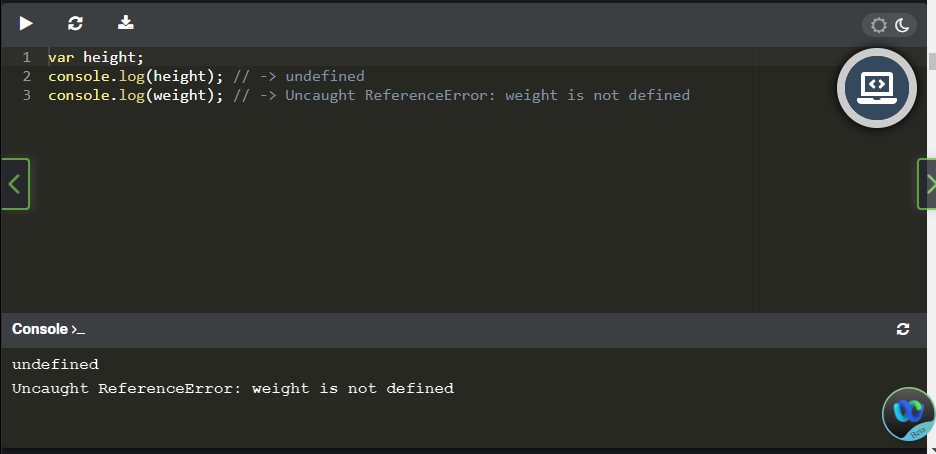
console.log(weight); *// -> Uncaught ReferenceError: weight is not defined*

The first line is the variable **declaration** (we can see the **var** keyword). This declaration means that the word height will be treated as the name of the container for certain values.



The declaration, like other JavaScript instructions, should end with a semicolon. In the second line, we try to write out the value of this variable (that is, what is in the container) on the console. Because we haven't put anything there yet, the result is undefined (the interpreter knows this variable, but it has no value yet – the value is undefined). In the next line, we try to print out the contents of the weight variable ... which we forgot to declare. This time, we will see **ReferenceError**. The JavaScript interpreter, which executes our program, has informed us that it doesn’t know a variable by this name (so the variable itself is undefined).

In the example from the editor below, we use the keyword **var**.



The alternative to it is the keyword **let**. We use both keywords in the same way. Both are meant for declaring variables, and both can be found in different examples on the Internet or in books. However, they are not exactly the same, and we’ll discuss the differences in their operation later in this chapter (even in several places).

The keyword **var** comes from the original JavaScript syntax, and the keyword **let** was introduced much later. Therefore, you will find var more in older programs. Currently, it is highly recommended to use the word **let** for reasons that we’ll discuss in a moment.

So, let's take a look at our example rewritten this time using the keyword **let.**

let height;

console.log(height); *// -> undefined*

One of the basic differences in the use of *var* and *let* is that let prevents us from declaring another variable with the same name (an error is generated). Using *var* allows you to re-declare a variable, which can potentially lead to errors in the program execution.

var height;

var height;

console.log(height); *// -> undefined*

The example above demonstrates the possibility of re-declaring a variable using the keyword var. In this situation, it will not cause an error, but in more complex programs a redeclaration, especially by accident, may no longer be without consequences. When declaring with let, the interpreter checks whether such a variable has already been declared, no matter if let or var is used in the previous declaration.

let height;

let height; *// -> Uncaught SyntaxError: Identifier 'height' has already been declared*

console.log(height);

So use **let** to declare variables, if only because you don't want to accidentally declare a variable again.

#### 2.0.4 Initializing variables

After a successful declaration, the variable should be **initialized**, in other words, it should be given its first value. **Initialization** is done by assigning a certain value to a variable (indicated by its name). To assign it, we use the operator **=**.



You can assign to a variable: a specific value; the contents of another variable; or, for example, the result returned by a function.

Initialization can be done either together with the declaration, or separately as an independent command. It is important to enter the first value into the variable before trying to read, modify, or display it.

let height = 180;

let anotherHeight = height;

let weight;

console.log(height); *// -> 180*

console.log(anotherHeight); *// -> 180*

weight = 70;

console.log(weight); *// -> 70*

In the above example (check it in the editor), the declarations of the variables height and anotherHeight are combined with their initialization, while the variable weight is declared and initialized separately. The height and weight variables are initialized by providing specific values (more precisely, a number), while the anotherHeight variable receives a value read from the height variable. The values of all the variables are displayed on the console.

By the way, pay attention to one thing. If you specify a variable name in console.log, the interpreter recognizes it and displays its value. If you put the same name in quotation marks, it will be treated as plain text, and displayed as such.

let height = 180;

console.log(height); *// -> 180*

console.log("height"); *// -> height*

#### 2.0.5 Declarations and strict mode

JavaScript had some major changes introduced in 2009 and 2015. Most of these changes extended the language syntax with new elements, but some of them concerned only the operation of the JavaScript interpreters. Often it was about clarifying the interpreters' behavior in potentially erroneous situations, such as in cases of variable initialization without any prior declaration.

Let's look at an example:

height = 180;

console.log(height); *// -> 180*

At first glance, you can see that we’ve forgotten to declare the variable height. The original JavaScript syntax allowed for such negligence, and at the moment of initialization it made this declaration for us. It seems like quite a good solution, but unfortunately it can sometimes lead to ambiguous and potentially erroneous situations (we’ll say a few more words about it while discussing the scope).

Let's modify our example:

"use strict";

height = 180; *// -> Uncaught ReferenceError: height is not defined*

console.log(height);

At the beginning of our code, we’ve added **"use strict";.** This statement has radically changed the behavior of the interpreter. Why? We use it when we want to force the interpreter to behave according to modern JavaScript standards. So, as long as you aren’t running some really old code, you should always use it. And this time, using a variable without its previous declaration is treated as an error.

The sentence **"use strict";** must be placed at the very beginning of the code. It will cause the interpreter to deal with the rest of the code using strict mode, which is the modern JavaScript standard. All further examples in our course will be prepared to work in this mode by default, even if **"use strict";** does not always appear at the beginning of the code.

#### 2.0.6 Changing variable values

Variables, as their name suggests, can store data that will vary. Changes are made by assigning a new value to the variable, which overwrites the previous one.

1 let steps = 100;

2 console.log(steps); *// -> 100*

3 steps = 120; *// -> 120*

4 console.log(steps);

5 steps = steps + 200;

6 console.log(steps); *// -> 320*

In our example, we’ve declared a variable called steps. Initially, it contains the number 100, which is then changed to 120. Then we add 200 to the current contents of the variable, as a result of which the variable contains 320.

Variables in the JavaScript language are untyped (or, to be more precise, they are weakly and dynamically typed). This means that JavaScript will not control what type of value we store in the variable. What exactly is the data type? You can probably intuitively answer this question yourself. The type determines the belonging of a given data to a certain set that share the same properties and on which you can perform the same operations. Data types vary greatly depending on the programming language. In JavaScript, the main types are number and character string. We will talk much more about types in the next chapter. Let's declare a few variables and initialize them with values of different types:

1let greeting = "Hello!";

2let counter = 100;

As you can see, the greeting variable is initiated with a value of the string type, while the counter variable is initiated with a value of the number type. Continuing the example, we will make a small change in the contents of the greeting variable.

1console.log(greeting); *// -> Hello!*

2greeting = 1;

3console.log(greeting); *// -> 1*

JavaScript allows us to easily replace the greeting variable with a value whose type is different from the one originally stored there. JavaScript goes one step further and not only allows us to change the types of values kept in a variable, but it also performs their implicit conversion if necessary (we will also return to this topic of conversion when discussing types). Let's restore the original value of the greeting variable and then add the value of the counter variable to it.

1greeting = "Hello!";

2greeting = greeting + counter;

3console.log(greeting); *// -> Hello!100*

The interpreter will check the type of value stored in the greeting variable and convert the value from the counter variable to the same type before performing an addition operation. As a result, the string **"100"** will be added to the **"Hello!"** character string and stored to the greeting variable. By the way, note that JavaScript interprets **100** as a number, but **"100"** as a string.

#### 2.0.7 Constants

The **const** keyword is used to declare containers similar to variables. Such containers are called **constants**. Constants are also used to store certain values, but once values have been entered into them during initialization, they can no longer be modified. This means that this type of container is simultaneously declared and initialized. For example, the following declaration of the greeting constant is correct:

1const greeting = "Hello!";

But this next one definitely causes an error:

1const greeting; *// -> Uncaught SyntaxError: Missing initializer in const declaration*

2greeting = "Hello!";

As we said, a change in the constant is impossible. This time the declaration is correct, but we try to modify the value stored in the constant.

1const greeting = "Hello!";

2greeting = "Hi!"; *// -> Uncaught TypeError: Assignment to*

*constant variable.*

The main purpose of a constant is to eradicate the possibility of accidentally changing a value stored in it. This is important when we have some values that really should never change. Typical examples of constants are paths to resources, tokens, and other data that never change throughout the lifetime of the script.

But constants can also be used as sub results in calculations or in other places where whatever information was gathered or calculated will not change any further. Using a const, besides preventing a value from being changed by mistake, allows the JavaScript engine to optimize the code, which may affect its performance.

#### 2.0.8 Scope

Until now, we assumed that after declaring a variable, its name could be used in the whole code of the program (i.e. the scope of the variable is global). This is not entirely true – the scope of a variable depends on where it is declared. Unfortunately, for a good understanding of variable scope, we need to learn a few more programming elements, such as conditional instructions or functions, which will be discussed in more detail later in the course. So here we will limit ourselves to basic information, and will come back to this issue in different parts of the course. One of the basic elements that influence the scope of variables is a **program block.**

**Program blocks**

We can separate the code of a program into blocks. In the blocks that we create using curly brackets, there is a set of instructions, which for some reason should be treated independently. The blocks are usually associated with conditional instructions, loops, or functions, which we will talk about later. We can also separate a block of a program unrelated to anything special, simply by choosing a certain range of instructions (in practice, this is not particularly justified, and for now we will only do it for educational reasons).

Let's look at an example:

1let counter;

2console.log(counter); *// -> undefined*

3{

4    counter = 1;

5    console.log(counter); *// -> 1*

6}

7counter = counter + 1;

8console.log(counter); *// -> 2*

First, we declare the variable counter. Then we open a block inside which we initialize this variable and display its contents. Outside the block, we increase the value stored in the variable by 1 and display it again. In this case, the interpreter will execute the program as if it hadn't noticed the block, going through the instructions before the block, in the block, and after the block. Creating a block here, without, for example, conditional instructions, has no real justification – it is just an example of using brackets.

Program blocks can be nested, that is, we can create one block inside of another one.

1let counter;

2console.log(counter); *// -> undefined*

3{

4    counter = 1;

5    {

6    console.log(counter); *// -> 1*

7    }

8}

9counter = counter + 1;

10console.log(counter); *// -> 2*

By the way, please note that the code inside the block has been moved to the right. This is called an **indentation**. For a JavaScript interpreter, it doesn't matter at all, but it definitely increases the readability of the code, allowing the readers (including you) to quickly find out which parts of the code are inside, and which are outside, the block. Code editors usually add indentations in the right places by themselves, but it is a good habit to remember this yourself and, if they do not appear automatically, format the code by hand.

**let and const**

It's time to move on to determine what is actually going on with these scopes. Unfortunately, the scopes of variables (and constants) declared with **let** and **const**look slightly different than those declared with var. So we will discuss them independently.

The first rule is simple. If we declare any variable or constant using **let** or **const**, respectively, outside the code blocks, they will be **global**. By this we mean that their names will be visible throughout the program, outside blocks, inside blocks, in functions, and so on. We will be able to refer to them anywhere by their names, and of course have access to their values.

What happens if we declare something using let or const inside a block? This will create a local variable or constant. It will be visible only inside the block in which it was declared and in blocks that can optionally be nested in it.

Let's look at a simple example:

1let height = 180;

2{

3    let weight = 70;

4    console.log(height); *// -> 180*

5    console.log(weight); *// -> 70*

6}

7console.log(height); *// -> 180*

8console.log(weight); *// -> Uncaught ReferenceError: weight is not defined*

The height variable, declared outside the block, is global. The weight variable is local – its scope is limited by the block in which it was declared. This is clearly visible when trying to display the values of both variables inside and outside the block. We can also test the case with nested blocks:

1let height = 200;

2{

3    let weight = 100;

4    {

5    let info = "tall";

6    console.log(height); *// -> 200*

7    console.log(weight); *// -> 100*

8    console.log(info); *// -> tall*

9    }

10    console.log(height); *// -> 200*

11    console.log(weight); *// -> 100*

12    console.log(info); *// -> Uncaught ReferenceError: info is not defined*

13   }

As you can see, the info variable declared in the most internal block is visible only inside it. The weight variable is visible both inside the block in which it was declared and inside the block nested in it. And the global variable height is visible everywhere.

Simple, isn't it?

**var**

In the case of variable declarations using the keyword **var**, the situation is slightly different. The variable declared using it outside the blocks will, as in the case of let, be global, in other words, it will be visible everywhere. If you declare it inside a block, then... well, it will usually turn out to be global again.

Let's start with a simple example:

1var height = 180;

2{

3    var weight = 70;

4    console.log(height); *// -> 180*

5    console.log(weight); *// -> 70*

6}

7console.log(height); *// -> 180*

8console.log(weight); *// -> 70*

As expected, both variables, **height** and **weight**, turn out to be global. Will the variables declared using **var** always, regardless of the place of declaration, be global? Definitely not. The problem is that var ignores ordinary program blocks, treating them as if they do not exist. So in what situation can we declare a local variable using var? Only inside a function. We will devote a lot of space to discussing the function, and then we will come back to the problem of the variable scope as well. Now we will try to present and discuss only a simple example, which will show that var variables are sometimes local, too.

#### 2.0.9 A brief word about functions

Let's start by explaining what **functions** are. It often happens that a certain piece of code, performing some specific task, will be used many times. Yes, you can copy this piece of code, all of its instructions, to any place where you want to use it. However, this would be very inefficient. First of all, the size of our program would grow unnecessarily. Secondly, if we would like to make some changes to this piece of code, for example, to correct some bug, we would have to do it in every place where we used it.

A simple solution to this problem is a function. A **function** is just a separated piece of code that you name, in the same way that you name a variable. If you want to use it somewhere, you simply refer to it using that name (we say that we *call* the function).

The declaration of a simple function, let's say **testFunction**, may look like this:

1function testFunction() {

2    console.log("Hello");

3    console.log("World");

4}

The way to define the function shown in the example is one of several available in JavaScript. The definition starts with the **function** keyword, followed by the function name we invented. After the name, you see parentheses, which optionally could contain parameters passed to the function (we will come back to this when we discuss the function more precisely). Then we open the program block, which contains the instructions belonging to the function. When defining a function, the instructions contained in the function are not executed. To execute the function, you must call it independently, using its name.

Take a look at the following program:

1console.log("let's begin:"); *// -> let's begin:*

2console.log("Hello"); *// -> Hello*

3console.log("World"); *// -> World*

4console.log("and again:"); *// -> and again:*

5console.log("Hello"); *// -> Hello*

6console.log("World"); *// -> World*

7console.log("and once more:"); *// -> and once more:*

8console.log("Hello"); *// -> Hello*

9console.log("World"); *// -> World*

It will print out a sequence of text on the console:

let's begin:

Hello

World

and again:

Hello

World

and once more:

Hello

World

#### 2.0.10 The var keyword - continued

After this short introduction to functions (this is obviously not our last meeting with them) let's return to the keyword **var** and variable scopes.

If we declare a variable using the keyword var inside a function, its scope will be limited only to the inside of that function (it's a local scope). This means that the variable name will be correctly recognized only inside this function.

Let's consider the following example:

1var globalGreeting = "Good ";

2

3function testFunction() {

4    var localGreeting = "Morning ";

5    console.log("function:");

6    console.log(globalGreeting);

7    console.log(localGreeting);

8}

9

10testFunction();

11

12console.log("main program:");

13console.log(globalGreeting);

14console.log(localGreeting); *// -> Uncaught ReferenceError: localGreeting is not defined*

First of all, run this program and observe the results on the console. What happened, and above all, why did it happen?

Let's take a closer look at the code. In the example, we declared the global variable **globalGreeting**. Then we defined the **testFunction** function, inside which we declared the local variable **localGreeting**. Then we called the **testFunction** function, which resulted in writing out the values of both variables (inside the function, we have access to both the global variable and the local ones). Attempting to access the local variable **localGreeting**outside the function will fail. So we’ve finally succeeded in demonstrating that variable declarations using the word **var**can also be local.

#### 2.0.11 Variable shadowing

JavaScript allows for variable shadowing. What does that mean? It means that we can declare a global variable and a local variable of the same name.

In the local scope, in which we declare a local variable using its name, we will have access to the local value (the global variable is hidden behind the local one, so we do not have access to it in this local scope). Using this name outside the local scope means that we will be referring to the global variable. **This is not best programming practice, however, and we should avoid such situations.** It is not difficult to guess that with a bit of inattention, using this mechanism can lead to unintended situations and probably to errors in the operation of the program.

If we are to avoid such situations, it would be good to see exactly what they are about. Let's begin with an example without shadowing:

1let counter = 100;

2console.log(counter); *// -> 100*

3{

4    counter = 200;

5    console.log(counter); *// -> 200*

6}

7console.log(counter); *// -> 200*

The **counter** variable, declared at the beginning of the program, is a global variable. Throughout the program, also inside the block, we operate on this very variable. A small change in the code is enough for the program to behave completely differently.

1let counter = 100;

2console.log(counter); *// -> 100*

3{

4    let counter = 200;

5    console.log(counter); *// -> 200*

6}

7console.log(counter); *// -> 100*

You see the difference? This time in the block, instead of **counter = 200;** (i.e. changes in the contents of the global counter variable), the **let counter = 200;** statement appears (i.e. declarations of the local variable combined with its initialization). The interpreter would consider such a situation to be wrong if the re-declaration appeared in the same scope.

However, the declaration is local (it’s a different scope than global) and all references to the variable with this name inside the block will refer to this local variable. Outside the block, the global variable will still be seen under the same name. Pay attention to the values displayed by the console.

Shadowing may not only be concerned with the situation in which a local variable covers a global variable. If nested scopes appear (e.g. nested blocks in the case of a let declaration), the local variable declared in a more nested block will overshadow the local variable of the same name declared in the external block.

Shadowing is also present in variable declarations using the word **var**, and this time the local scope is limited not by the program block, but by the function block.

1var counter = 100;

2

3function testFunction() {

4    var counter = 200;

5    console.log(counter);

6}

7

8console.log(counter); *// -> 100*

9testFunction(); *// -> 200*

10console.log(counter); *// -> 100*

In most cases, this is not desirable, so try to avoid giving the same variable names to multiple variables, regardless of where you declare them.

#### 2.0.12 Hoisting

Remember how we said that all variables must be declared before use? This is not entirely true, and really the word "should" is a better fit than "must". Of course, good practice is always to declare variables before they are used. And stick to this. But the original JavaScript syntax allows for some deviations from this rule.

The JavaScript interpreter scans the program before running it, looking for errors in its syntax, among other things. It does one more thing on this occasion. It searches for all variable declarations and moves them to the beginning of the range in which they were declared (to the beginning of the program if they are global, to the beginning of the block if it is a local **let** declaration, or to the beginning of the function if it is a local **var** declaration). All this happens, of course, in the interpreter memory, and the changes are not visible in the code.

**Hoisting**, because we are talking about it, is a rather complex and frankly speaking quite incoherent mechanism. Understanding it well requires the ability to freely use many JavaScript elements, which we have not even mentioned yet.

What's more, it's rather a curiosity than something practical that you will use when writing programs, so we will look at just one small example that will allow us to roughly understand the very principle of hoisting. This may make it easier for you to understand some surprising situations when writing your own code, or testing examples you find in various sources.

1var height = 180;

2console.log(height); *// -> 180*

3console.log(weight); *// -> Uncaught ReferenceError: weight is not defined*

In the above example, we forgot to declare the variable weight. The result is obvious: we’re referring to a variable (that is, we’re trying to read its contents) which does not exist. Something like this must end in an error.

Let's make a small change:

1var height = 180;

2console.log(height); *// -> 180*

3console.log(weight); *// -> undefined*

4var weight = 70;

5console.log(weight); *// -> 70*

This time we declared our variable, but in a rather strange place. Together with the declaration, we also performed initialization. The result of the program may be a bit surprising. This time there are no errors. Hoisting has worked, and the declaration has been moved by the interpreter to the beginning of the range (in this case the program).

However, the attempt to display the contents of the weight variable give two different results. Why? Hoisting only concerns the declaration, not initialization. So the value 70, which we assign to the weight variable, remains on the line where the original declaration is. The above example is interpreted by the JavaScript engine more or less in the following way:

1var weight;

2var height = 180;

3console.log(height); *// -> 180*

4console.log(weight); *// -> undefined*

5weight = 70;

6console.log(weight); *// -> 70*

Hoisting unfortunately works a little differently with the **let** and **const** declarations.

However, we will not go into it. It is enough that you are aware of the phenomenon.

And most of all, you will remember ALWAYS to declare variables before using them.

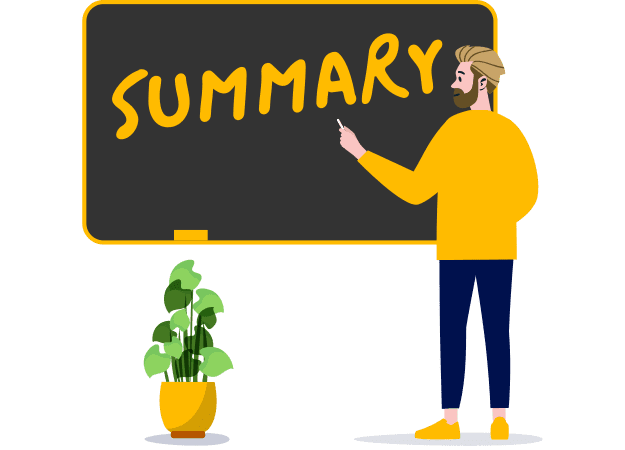
#### 2.0.13 SECTION SUMMARY

Using variables, in other words, declaring, initializing, changing, or reading their values is an elementary part of practically every programming language.

JavaScript is no exception, as you need to use variables to program in it.

Remember to declare variables before using them. Pay attention to where you declare them – whether they are local or global.

Try to use the keywords **let** and **const**, not the word **var**.



Knowing the latter will be useful not for understanding the examples found in various sources, but rather so that you can avoid using it yourself.

Remember not to use the same names for different variables, even if you declare them in different ranges.

And, of course, give the variables names that will be related to what you want to store in them – the code should be readable not only to the interpreter, but also to people.

#### 2.0.14 SECTION QUIZ

**Question 1:** Let's play florist. Declare six variables, remembering to name them according to their purpose:

* the price of a single rose (8) and the number of roses you have (70)
* the price of a single lily (10) and the number of lilies you have (50)
* the price of a single tulip (2) and the number of tulips you have (120)

Now declare three variables, one each for the roses, lilies, and tulips you have, in which you place their total price. Insert the corresponding values into the variables using the variables declared in the previous step. Finally, declare a variable in which you store the price of all your flowers (again, use the previous variables for initialization). Display all inventory information in the console in the following form:

Rose – unit price: 8 , quantity: 70 , value: 560

Lily – unit price: 10 , quantity: 50 , value: 500

Tulip – unit price: 2 , quantity: 120 , value: 240

Total: 1300

Check

**Question 2:** Modify the code from the previous example.

Assume that the prices of flowers will be constant (they will not change). Declare and initialize the remaining variables in the same way as in the previous example. Display all the gathered information in the console. Now decrease the number of roses by 20 and lilies by 30.

Display all the collected information in the console again.

Check