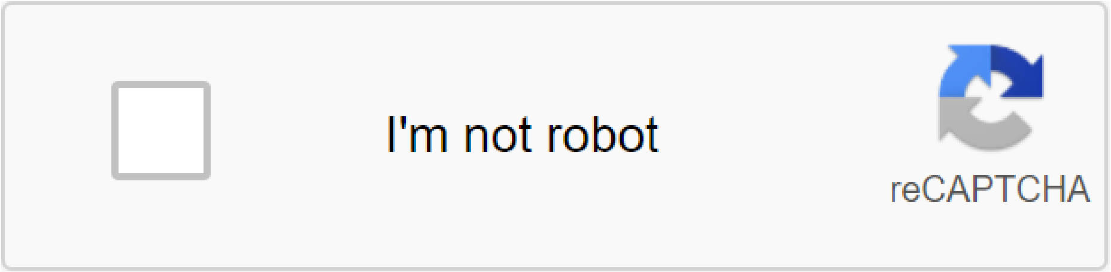
Programming javascript applications pdf español



Continue

Start learning here by typing your name surrounded by quotation marks and ending with a semicolon. For example, you can type jamie's name; and then press enter. Hackr.io es una comunidad para encontrar y compartir los mejores cursos y tutoriales en línea. Únete a ellos, solo lleva 30 segundos. Eloquent Javscript en Español January 17, 2016 Jul 11, 2015 JavaScript is one of the most popular programming languages in the world. I think it's a great choice for your first programming language ever. We mainly use JavaScript to create web applications for web server applications using Node.js but JavaScript is not limited to these things and can also be used to create mobile applications using tools such as React Native to create programs for microcontrollers and the Internet of Things to create smartwatch apps Can basically do nothing. It's so popular that everything new that comes up will have some JavaScript integration at some point. JavaScript is a high-level programming language: it provides abstractions that allow you to ignore the details of the computer on which it is running. It manages memory automatically by using a garbage collector so that you can focus on your code instead of managing memory, such as other languages such as C, and provides multiple constructs that allow you to deal with very advanced variables and objects. dynamic: Unlike static programming languages, a dynamic language performs many things at run time that a static language does at compile time. This has pros and cons, and gives us powerful features such as dynamic writing, late binding, reflection, functional programming, object run time change, shutdown, and much more. Don't worry if these things are unknown to you - you'll know them all by the end of the course. dynamically typed: The variable does not enforce the type. You can reasscribe any type to a variable, for example, by assigning an integer to a variable that contains a string. loosely typed: Unlike strong typing, loosely (or poorly) typed languages do not enforce object type, which allows for greater flexibility, but deny us the security of typing and type checking (something that TypeScript - which is based on JavaScript - provides) interpreted: it is commonly known as interpreted language, which means it doesn't need a build step before the program can run, unlike C , Java or Go for example. In practice, browsers compile JavaScript before executing it, for performance reasons, but this is invisible to you - there is no additional step. multi-paradigm: the language does not enforce any specific programming paradigm, unlike java for example, which forces the use of programming or C, which forces imperative programming. JavaScript can be used to write using the object paradigm, using prototypes and new new ES6). You can write JavaScript in a functional programming style, with its first-class features, and even in an imperative (c-like) style. In case you're wondering, JavaScript has nothing to do with Java, it's a bad name choice, but we have to live with it. Update: You can now get a PDF version and ePub of this JavaScript Beginner's Guide.A little history Created in 1995, JavaScript has come a very long way from humble beginnings. It was the first scripting language that was supported on a national basis by web browsers, and thus gained a competitive advantage over any other language, and today it is still the only scripting language we can use to build web applications. There are other languages, but they all need to be compiled into JavaScript - or more recently to WebAssembly, but this is a different story. At first JavaScript was not nearly as powerful as it is today, and was mainly used for fancy animations and a miracle known at the time as Dynamic HTML. With the growing needs demanded by the web platform (and still requires), JavaScript was also required to evolve to meet the needs of one of the world's most widely used ecosystems. JavaScript is now widely used outside the browser. Node.js growth over the past few years has unlocked backend development, following java domains, Ruby, Python, PHP, and more traditional server-side languages. JavaScript is now also the language that powers databases and many other applications, and it's even possible to create built-in apps, mobile apps, TV apps, and more. What started out as a small language inside the browser is now the most popular language in the world. JavaScript only Sometimes it is difficult to separate JavaScript from the functionality of the environment in which it is used. For example, the console.log() line, which can be found in many code examples, is not javascript. Instead, it is part of a huge library of APIs provided to us in the browser. In the same way, it can sometimes be difficult on the server to separate JavaScript features from the APIs provided by node.js. Is the feature provided by React or Vue? Or is it plain JavaScript or vanilla JavaScript, as it is often called? In this book I'm talking about JavaScript, language. Without complicating the learning process with things that are beyond it, and delivered by external ecosystems. A brief introduction to JavaScript syntax In this small introduction, I want to talk about 5 concepts: literal literal sensitivity in case of attention interval White JavaScript does not consider white space to be significant. Spaces and line breaks can be added in any way, at least in theory. In practice, you will most likely maintain a well-defined style and apply to what people often use, and you will force it with a linter or style tool like Prettier. For For I always use 2 characters of space for each indentation. JavaScript case-sensitive is distinguished. A variable called something is different than something. The same applies to any identifier. Literals Define a literal as a value written in the source code, such as a number, string, logical string, and more advanced constructs such as object literals or array literals: 5 'Test' true ['a', 'b'] {color: 'red', shape: 'Rectangle'} Identifiers Identifiers identifier is a sequence of characters that can be used to identify a variable, function, or object. It can start with a letter, dollar sign $ or underscore _, and can contain numbers. Using Unicode, a letter can be any allowed character, for example, an emoji 😊. Test test _test Test1 $test Dollar sign is commonly used to refer to DOM elements. Some names are reserved for internal JavaScript use and cannot be used as identifiers. Comments are one of the most important parts of any program, in any programming language. These are important because they allow us to describe the code and add important information that would otherwise not be available to other people (or us) reading the code. In JavaScript, we can write a comment on a single line using //. Everything after // is not considered code by the JavaScript interpreter. Like this: // true comment //another comment Another comment type is a multiline comment. It starts with /* and ends with */. Everything in between is not treated as code: /* some comment */ Semicolons Each line in JavaScript is optionally terminated with semicolons. I said optionally because the JavaScript interpreter is smart enough to enter semicolons for you. In most cases, you can skip semicolons completely from programs without even thinking about it. This fact is very controversial. Some developers will always use semicolons, others will never use semicolons, and you'll always find code that uses semicolons and code that doesn't. My personal preference is to avoid semicolons, so my examples in the book won't include them. The value of the Greeting String is a value. A number like 12 is a value. hello and 12 are values. string and number are the types of these values. A type is a type of value, its category. We have many different types in JavaScript, and we'll talk about them in detail later. Each type has its own characteristics. When we need to have a reference to a value, we assign it to a variable. A variable can have a name, and the value is stored in a variable, so that we can later access that value by using the variable name. Variable variables are a value assigned to an identifier so that you can reference and use it later in the program. This is because JavaScript is loosely you often hear about. The variable must be declared before it can be used. We have 2 main main to declare variables. The first is to use const: const a = 0 The second way is to use let: let a = 0 What is the difference? const defines a constant reference to a value. This means that you cannot change the reference. You cannot reasscribe a new value to it. You can use let to assign a new value to it. For example, you cannot do this: const a = 0 a = 1 Because you get the error: TypeError: Assignment to a constant variable.. On the other hand, you can do this by using let: let a = 0 a = 1 const does not mean constant in the way some other languages like C mean. In particular, this does not mean that the value cannot be changed - it means that it cannot be reasscribeed. If a variable points to an object or array (we'll see more about objects and arrays later), the contents of the object or array can change freely. const variables must be initialized at declaration time: const a = 0, but let the values be initialized later: let a = 0 You can declare multiple variables at once in the same statement: const a = 1, b = 2 let c = 1, d = 2 But you cannot re-declare the same variable more than once: let = 1 let a = 2 or you get a duplicate error declaration. My advice is to always use const and use only let it when you know that you need to reasscribe a value to this variable. Why? Because the less power our code has, the better. If we know that the value cannot be reassigned, this is one less source of errors. Now that we have seen how to work with const and let, I want to mention var. Until 2015, var was the only way we could declare a variable in JavaScript. Today, a modern codebase will most likely just use const and let it. There are some basic differences that detail in this post, but if you're just getting started, you may not care about them. Just use const and let. JavaScript variable types do not have any type attached. They are of no type. After you assign a value with some type to a variable, you can later reassign the variable to host another type of value without any problems. In JavaScript, we have two main types: primitive types and object types. Primitive types Primitive types are numbers of logical symbol strings and two special types: null and undefined. Object types Any value that is not of the primitive type (string, number, boolean, null, or undefined) is an object. Object types have properties and also have methods that can act on those properties. We'll talk more about the facilities later. An expression expression is a single unit of JavaScript code that the JavaScript engine can evaluate and return a value. Expressions can vary in complexity. We start with very simple, called base expressions: 2 0.02 something real false to //the current scope i//where i is a variable or a constant Arithmetic expressions are expressions that take a variable and an operator (more on operators soon), and result in a number: 1 / 2 i++ i -= 2 2 * 2 String expressions are expressions that cause the string: 'A ' + 'string' Logical expressions use logical operators and recognize the boolean value: a &amp;&amp;amp; b a || b !a More advanced expressions include objects, functions, and arrays, and I'll introduce them later. Operator operators allow you to obtain two simple expressions and combine them to create a more complex expression. We can classify operators based on the operands they work with. Some operators work with 1 operand. Most work with 2 operands. Only one operator works with 3 operands. In this first introduction to operators we will introduce the operators that are most likely to be known: operators with 2 operands. I have already introduced one when we talk about variables: assignment operator =. You are using = assign a value to a variable: let b = 2 Let's now enter another set of binary operators that are already familiar with basic mathematics. const three = 1 + 2 const four = three + 1 operator + no join strings also if you use strings, so note: const three = 1 + 2 three + 1 // 4 three + 1 // three1 const two = 4 - 2 Split operator (/) Returns the quotient of the first operator and the second: const result = 20 / 5 ///result === 4 const result = 20 / 7 //result === 2.857142857142857 If you divide by zero, JavaScript does not raise any error, but returns Infinity (or -Infinity if the value is negative). 1 / 0 //Infinity -1 / 0 //-Infinity Remaining operator (%) The remainder is a very useful calculation in many applications: const result = 20 % 5 //result === 0 const result = 20 % 7 //result === 6 And the rest by zero is always NaN, a special value means No number: 1 % 0 //NaN -1 % 0 //NaN Multiplication operator (*) two multiplication numbers 1 * 1 2 / 2 -1 * 2 //-2 Exponential operator (**) Lift the first operand to the power of the second operand 1 ** 2 //1 2 ** 1 //2 2 ** 2 //4 2 ** 8 //256 8 ** 2 //64 Priority rules Each statement made with multiple operators on the same line introduces problems. Let's take this example: let = 1 * 2 + 5 / 2 % 2 The score is 2.5, but why? What operations are performed first and which operations do I have to wait for? Some operations take precedence over others. Precedence rules are listed in this table: Description of operator * / % multiplication/division + - addition/deejuncing = assignment Operations at the same level (such as + and -) are performed in the order in which they were found, from left to right. In accordance with these principles, the above operation can be solved as follows: let a = 1 * 2 + 5 / 2 % 2 let = 2 + 5 / 2 % 2 let a = 2 + 2,5 % 2 let a = 2 + 0,5 let = 2,5 comparison operators After assignment and mathematical operators, the third set of operators that I want to enter is a conditional operator. Using the following you can compare two numbers or two strings. Comparison Comparison Comparison always returns a boolean value, a value that is true or false.) These are the disqualified comparison operators: &lt; means less than &lt;= means less than or equal to &gt; means greater than &gt;= means greater than or equal to Example: let = 2 &gt; = 1 //true In addition to those, we have 4 equality operators. They accept two values and return a boolean value: === checks for equality !== checks inequality Note that we also have == and != in JavaScript, but I strongly suggest using only === and !== because they can prevent some subtle problems. Conditionality Thanks to comparison operators, we can talk about conditions. The if statement is used for the program to take a route or other, depending on the result of the evaluation expression. This is the simplest example that always executes: if (true) { //do something } on the contrary, it is never performed: if (false) { //do something (? never ?) } The condition checks the expression that you pass to it for true or false. If you pass a number that is always rated as true, unless it's 0. If you pass a string, it is always evaluated as true unless it is an empty string. These are general rules for throwing types to a boolean value. Have you noticed curly braces? This is called a block and is used to group a list of different statements. You can place a block wherever you can have one statement. And if you have one statement to execute after conditionality, you can skip the block, and just write the statement: if (true) doSomething() But I always like to use braces to be more clear. You can provide the second part to the if statement: else. Include the statement to be executed if the if condition is false: if (true) { //do something } else { //do something else } Because it otherwise accepts the statement, you can nest another if/else statement in it: if (a === true) { //do something } else if (b=== true) { //do something else } else { //fallback } Arrays Array is a collection of elements. Arrays in JavaScript are not type on their own. Arrays are objects. We can initialize an empty array in these 2 different ways: const a = [] const a = Array() The first one uses the array literal syntax. The second uses the built-in Array function. The array can be pre-populated with this syntax: const a = [1, 2, 3] const a = Array.of(1, 2, 3) The array can contain any value, even values of different types: const a = [1, 'Flavio', ['a', 'b']] Because we can add an array to the array, we can create multidimensional arrays, which have very useful applications (e.g. matrix): consure matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] matrix[0][0] //1 matrix[2][2][0]0] //7 You can access any element of the array by referring to its index, which starts from zero: a [0] //1 a[1] //2 a[2] //3 You can initialize a new array with using this syntax, which first initials an array of 12 elements and fills in the element number 0: Array(12).fill(0) You can obtain the number of elements in an array by checking its length property: const a = [1, 2, 3] a.length //3 Note that you can set the length of the array. If you assign a larger number than the arrays of the current capacity, nothing happens. If you assign a smaller number, the array is cut at this position: const a = [1, 2, 3] and //[1, 2, 3] a.length = 2 a //[1, 2] How to add an element to an array We can add an element at the end of an array using push(): a.push(4) We can add an element at the beginning of the array using the unshift method (): a.unshift(0) a.unshift(-2, -1) We can remove an element from the end of the array using the pop(): a.pop() method We can remove the element from the beginning of the array using the shift(): a.shift() Method As connect two or more arrays You can join multiple arrays by using concat(): const a = [1 , 2] const b = [3, 4] const c = a.concat(b) //[1,2,3,4] a//[1,2] b/You can also use the spread operator (...) this way: const a = [1, 2] const b = [3 , 4] const c = [... Until... b] c //[1,2,3,4] How to find a specific element in an array You can use the find() method of the array: a.find((element, index, array) =&gt; { //return true or false }) Returns the first element that returns true and returns undefined if the element is not found. Commonly used syntax is: a.find(x =&gt; x.id === my_id) The above line will return the first element in the array, which has the identifier === my_id. findIndex() works similarly to find(), but returns the index of the first element that returns true, and if not found, returns undefined: a.findIndex((element, index, array) =&gt; { //return true or false }) Another method is includes(): a.includes(value) Returns true if it contains a value. a.includes(value, i) Returns true if the value contains after i. String string strings is a sequence of characters. It can also be defined as a string literal that is enclosed in quotation marks or quotation marks: Another string I personally prefer single quotation marks at all times and I use double quotation marks only in HTML to define attributes. You can assign a string value to a variable this way: name const = 'Flavio' You can specify the length of a string using its length property: 'Flavio'.length //6 const name = 'Flavio' name.length //6 This is an empty string: ''. Its length property is 0: ''.length //0 Two strings can be combined using the + : A + string operator + can be used to interpolate variables: const name = 'Flavio' My name is + name //My name is Flavio Another way to define strings is to use template literals defined inside backticks. They are especially useful to make multiline strings much simpler. For single or double quotation marks, it is not easy to define a multiline string - use fleeing characters. When you open a literal template with a backtick, just press enter enter create a new line without special characters, and it is rendered as-is: const string = Hey this string is awesome!, template literals are also great because they provide an easy way to interpolate variables and expressions in strings. You can add anything, even expressions: const string = 'something ${1 + 2 + 3}' const string2 = 'something ${foo() ? 'x' : 'y'}' Loops are one of the main control structures of JavaScript. With a loop, we can automate and repeat a block of code, but many times we want to run it, even indefinitely. JavaScript provides multiple ways to iterate through loops. I want to focus on 3 ways: during the loop for the loop for.. while the while loop is the simplest loop structure that JavaScript provides us with. We add a condition after the while keyword and provide a block that runs until the condition is evaluated as true. Example: const list = ['a', 'b', 'c'] let i = 0 while (i &lt; list.length) { console.log(list[i]) //value console.log(i) //index i = and + 1 } You can break the loop while using the break keyword, yes: while (true) { if (somethingIsTrue) break } and if you decide that in the middle of a loop you want to skip the current iteration, you can jump to the next iteration using continue: while (true) { if (somethingIsTrue) continue //do something else } Very similar to while, we have do.. during the loop. It is basically the same as while, except the condition is evaluated when a block of code is executed. This means that the block is always executed at least once. For example, const list = ['a', 'b', 'c'] let i = 0 to { console.log(list[i]) //value console.log(i) //index i = and + 1 }while (i &lt; list.length) for the second very important javascript loop structure is a loop. We use for keyword and pass a set of 3 statements: initialization, condition, and part of increment. For example, const list = ['a', 'b', 'c'] for (let i = 0; and &lt; list.length; i++) { console.log(list[i]) //value console.log(i) //index } As with while loops, you can break the for loop with a break and you can quickly move to the next iteration of the loop for the loop with continue. For... this loop is relatively recent (introduced in 2015) and is a simplified version for loop: const list = ['a', 'b', 'c'] for (const value of list) { console.log(value) //value } Functions In any moderately complex JavaScript program, everything happens inside the function. Features are a fundamental, essential part of JavaScript. What is a feature? A function is a block of code contained in itself. Here's a function declaration: getData() { // do something } The function can be run at any time by calling it, just like this: getData() The function can have one or more getData() { //do something } } getData(color) { //do something } getData function(color, age) { //to something } When we pass an argument, we call a function that passs parameters: getData function(color, age) { //do something } getData('green', 24) getData('black') Note that in the second call I passed the parameter black string as a color argument, but not age. In this case, the age inside the function is undefined. We can check that the value is not undefined using this conditional: getData(color, age) { //do something if (typeof age !== 'undefined') { //... } } typeof is an unambiguous operator that allows us to check the type of variable. You can also check this way: getData function(color, age) { //do something if (age) { //... } Although the condition will also be true if age is null, 0, or an empty string. You can have default values for parameters where they are not passed: getData(color = 'black', age = 25) { //do something } You can pass any value as a parameter: numbers, strings, logical values, arrays, objects, as well as functions. The function has a return value. By default, the function returns undefined unless you add a return keyword with the value: getData() { // do something returns 'hi!' } We can assign this return value to a variable when calling a function: getData() { // do something to return 'hi!' } the result let = getData() the result now has a string with hi! Value. Only one value can be returned. To return multiple values, you can return an object, or an array, like this: getData() { return ['Flavio', 37] } let [name, age] = getData() Functions can be defined inside other functions: const getData = () =&gt; { const dosomething = () =&gt; {} dosomething() return 'test' } The nested function cannot be called from outside the surrounding function. You can also return a function from a function. Arrow functions Arrow functions are a recent introduction to JavaScript. They are very often used instead of the regular features that I described in the previous chapter. You will find both forms used everywhere. Visually, they allow you to save functions with shorter syntax, from: getData() { //... } to () =&gt; { //... } But.. note that we don't have a name here. The arrow functions are anonymous. We need to assign them to a variable. We can assign a regular function to a variable, such as: let getData = getData() { //... } When we do this, we can remove the name from the function: let getData = function() { //... } and call the function with the variable name: let getData = function() { //... } getData() Same, what we do with the arrow functions: letData = () =&gt; { //... } getData() If the function body contains only a single statement, you can omit the parentheses and write everything in one line: const getData = () =&gt; console.log('hi!') The parameters are passed in parentheses: = (param1, param2) =&gt; console.log(param1, param2) If you have one (and only one) parameter, you can skip the parentheses completely: const getData = param =&gt; console.log(param) Arrow functions allow you to have an implicit return - values are returned without having to use the return keyword. Works when there is a single-line statement in the function body: const getData = () =&gt; 'test' getData() //'test' As with regular functions, we can have default parameter values in case they are not passed: const getData = (color = 'black', age = 2) =&gt; { //do something } And like regular functions, we can return only one value. Arrow functions can also include other arrow functions and even regular functions. The two types of functions are very similar, so you might ask why you introduced arrow functions. A big difference in regular functions is when they are used as object methods. This is something we will soon look at. Objects Any values that are not of the original type (string, number, boolean, symbol, null, or undefined) is an object. Here's how we define an object: const car = { } This is the literal syntax of an object, which is one of the most beautiful things in JavaScript. You can also use the new object syntax: const car = new Object() Another syntax is to use Object.create(): const car = Object.create() You can also initialize an object with a new keyword before a high-letter function. This function serves as a constructor for this object. There we can initiate arguments that we receive as parameters to configure the initial state of the object: function Car(brand, model) { this.brand = brand this.model = model } We initiate a new object with: const myCar = new Car ('Ford', 'Fiesta') myCar.brand //'Ford' myCar.model //'Fiesta' Objects are always passed by reference. If you assign a variable the same value to another, if it is a primitive type, such as a number or string, they are passed by value: Let's take this example: let's age = 36 let myAge = age myAge = 37 age //36 const car = { color: 'blue' } const anotherCar = carCar another.color = 'yellow' car.color //'yellow' Even arrays or functions are under the hood, objects, so it is very important to understand how they work. Object properties Objects have properties that consist of a label associated with a value. The value of a property can be of any type, which means that it can be an array, a function, or even an object, because objects can nest other objects. This is the literal syntax of the object we saw in the previous chapter: const car = { } We can define the color property this way: const car = { color: 'blue' } Here we have a car object with a property called color, with a blue value. Labels can be any string, but on special characters - if I wanted to include a non-valid character as a variable name in a property name, I would have to use quotation marks around it: it: car = { color: 'blue', 'the color': 'blue' } Invalid variable name characters include spaces, hyphens, and other special characters. As you can see, when we have multiple properties, we separate each property with a comma. You can retrieve the value of a property by using 2 different syntaxes. The first is the notation period: car.color //'blue' The second (which is the only one we can use for properties with invalid names), is the use of square brackets: car['the color'] //'blue' If you access a non-existent property, you will get an undefined value: car.brand //undefined As mentioned earlier, objects can have nested objects as properties: const car = { brand: { name: 'Ford' }, color: 'blue' } In this example, you can access the brand name using car.brand.name or car['brand']['name'] You can set the property value when defining an object. But you can always update it later: const car = { color: 'blue' } car.color = 'yellow' car['color'] = 'red' And you can also add new properties to the object: car.model = 'Fiesta' car.model //'Fiesta' Given the const car object = { color: 'blue', brand: Ford } you can remove the property from this object using delete car.brand Object Methods I spoke about the functions in the previous chapter. Functions can be assigned to function properties, and in this case they are called methods. In this example, the start property is assigned a function, and we can call it using the period syntax we used for the property, with parentheses at the end: const car = { brand: 'Ford', model: 'Fiesta', start: function() { console.log('Started') } } car.start() Inside a method defined by the function syntax() {} we have access to the instance of the object by referring to it. In the following example, we have access to brand and model property values using this.brand and this.model: const car = { brand: 'Ford', model: 'Fiesta', start: function() { console.log('Started ${this.brand} ${this.model}') } car.start() It is important to remember that distinction between regular functions and arrow functions - we do not have access to this if we use the arrow function: const car = { brand: 'Ford', model: 'Fiesta', start: () =&gt; { console.log(Started ${this.brand} ${this.model}') //not going to work } } car.start() This is because the arrow functions are not related to the object. This is the reason why regular functions are often used as object methods. Methods can accept parameters such as regular functions: const car = { brand: 'Ford', model: 'Fiesta', goTo: function(destination) { console.log('Going to ${destination}') } access .goTo('Rome') Class We talked about objects that are one of the most interesting parts of JavaScript. In this chapter we will go one level by introducing classes. What are classes? They are a way of defining pattern for multiple objects. Take person object: const person = { name: 'Flavio' } We can create a class class A person (note capital P, convention when using classes) who has the property name: class Person { name } Now from this class we initialize the flavio object this way: const flavio = new Person() flavio is called an instance of the person class. We can set the property value name: flavio.name = Flavio, and we can access it using flavio.name we do for object properties. Classes can store properties such as name and methods. Methods are defined in this way: Person class { hello() { return 'Hello, I am Flavio' } } and we can call methods in a class instance: Class Person { hello() { return 'Hello, I am Flavio' } } } const flavio = new Person() flavio.hello() There is a special method called constructor() that we can use to initialize class properties when creating a new instance of an object. This works this way: Person class { constructor(name) { this.name = name } hello() { return 'Hello, I am ' + this.name + '.' } Note how we use this access to the instance of the object. Now we can create an instance of a new object from the class, pass in a string, and when we call hello we will get a personalized message: const flavio = new Person('flavio') flavio.hello() //'Hello, I am flavio.' When an object is initialized, the constructor method is called with any passed parameters. Typically, methods are defined in an instance of an object, not in a class. A method can be defined as static to allow it to execute in a class instead of: Class Person { static genericHello() { return 'Hello' } } Person.genericHello() //Hello This is very useful sometimes. Class A inheritance can extend another class, and objects initialized with that class inherit all methods of both classes. Suppose we have a Person class: Person class { hello() { return 'Hello, I am a Person' } } } We can define a new class, a programmer that extends Person: class Programmer extends Person { } Now if a new object with a class programmer occurs, has access to hello() method: const flavio = new Programmer() flavio.hello() //'Hello, I'm a person' Inside the child class, you can refer to the parent class by calling super(): class Programmer extends Person { hello() { return super.hello() + I'm also a programmer.' } } const flavio = new Programr() flavio.hello() The above program prints Hello, I'm a person. I'm also a programmer.. Asynchonic programming and callbacks In most cases, JavaScript runs synchronously. That is, the line of code is executed, then the next one is executed, and so on. Everything is as you expect and how it works in most programming languages. However, there are times when you can't just wait for a line of code to execute. You can't just wait 2 seconds for a large file to load and stop the program. You can't just wait for a network resource to download before doing something else. JavaScript solves this problem using callbacks. One of the simplest examples of using callbacks is with timers. Timers are not part of JavaScript, but are provided by the browser and Node.js. Let me talk about one of the timers we have: setTimeout(). The setTimeout() function accepts 2 arguments: function and number. The number is milliseconds that must pass before the function is started. Example: setTimeout(() =&gt; { // runs after 2 seconds console.log('inside the function') }, 2000) A function containing the console.log('inside function') line will be executed after 2 seconds. If you add console.log('before'), and console.log('after') after the function before the function: console.log('before') setTimeout(() =&gt; { // works after 2 seconds console.log('inside the function') }, 2000) console.log('after') You will see this happening in the console: before inside the function The callback function is performed asynchronously. This is a very common pattern when working with a file system, network, events, or home in a browser. All the things I mentioned are not the core of JavaScript, so they are not explained in this manual, but you will find many examples in my other manuals available in . Here's how we implement callbacks in our code. We define a function that accepts the callback parameter, which is a function. When the code is ready to call back, we call it, passing the result: const doSomething = callback =&gt; { //do things //do things const result = /* .. */ callback(result) } Code using this function would use it as follows: doSomething(result =&gt; { console.log(result) }) Promises are an alternative way to deal with asynchronous code. As we saw in the previous chapter, with callbacks we will pass the function to another function call that will be called when the function processing is complete. Yes: doSomething(result =&gt; { console.log(result) }) When the doSomething() code is complete, it calls the function received as a parameter: const doSomething = callback =&gt; { //do things //do things const result = /* .. */ callback(result) } The main problem with this approach is that if we need to use the result of this function in the rest of our code, all our code must be nested inside the callback, and if we need to make 2-3 callbacks we introduce into what is usually defined a callback hell with multiple levels of functions indented into other functions: doSomething (result =&gt; doSomethingElse(anotherResult =&gt; { doSomethingElseAgain(yetAnotherResult =&gt; { console.log(result) }) }) }) }) Promises are one way to deal with this. Instead of doing: doSomething(result =&gt; { console.log(result) }) This is how we call a promise-based function: doSomething() .then(result =&gt; { }) First we call the function, and then we have the then() method that is called when the function ends. Indenting does not matter, it matters, you often use this style for clarity. Errors are often detected using the catch(): doSomething() .then(result =&gt; { console.log(result) }) .catch(error =&gt; { console.log(error) }) Now, to be able to use this syntax, the implementation of doSomething() must be a little special. Must use the Promises API. Instead of declaring it as a normal function: const doSomething = () =&gt; { } We declare it as a promise object: const doSomething = new Promise() and pass the function in the Promise constructor: const doSomething = new Promise() =&gt; { }) This function receives 2 parameters. The first is the function that we are calling for the promise to be resolved, the second is the function that we are calling for the promise to be rejected. const doSomething = new promise((resolve, reject) =&gt; { }) The promise solution means that the promise is successfully completed (resulting in the then() method being called in whatever it uses). Rejecting a promise means ending it with an error (which causes the catch() method to be called in any use). Here's how: const doSomething = new promise((resolve, reject) =&gt; { //some code const success = /* ... */ if (success) { resolve('ok') } else { reject('this error occurred') } }) We can pass a parameter to the recognition and rejection function of any type. The Async and Await Async functions are an abstraction at a higher level of promise. The asynchronous function returns a promise as in this example: const getData = () =&gt; { return new Promise((resolve, reject) =&gt; { setTimeout(() =&gt; resolve('some data'), 2000} Any code that wants to use this function will use the await keyword just before the function: const data = await getData() and thus all data returned by the promise will be assigned to the data variable. In our case, the data is a string of some data. With one specific caveat: every time we use the await keyword, we must do so inside a function defined as async. Yes: const doSomething = async () =&gt; { const data = await getData() console.log(data) } Asynchization duo/await allows us to have cleaner code and a simple mental model to work with asynchronous code. As you can see in the example above, our code looks very simple. Compare it to code using promises or callback functions. And this is a very simple example, the main benefits will come when the code is much more complex. For example, here's how you can get a JSON resource using the download API, and parse it with promises: const getFirstUserData = () = &gt; { // get users list return fetch('/users.json') // parse JSON .then(response =&gt; response.json()) // pick first user .then(users =&gt; users[0]) // get user =&gt; fetch('/users/${user.name}')/get user =&gt; fetch('/users/${user.name}'))() // analyze JSON .then(userResponse =&gt; response.json()) } getFirstUserData() And here's the same podana przy użyciu await/async: const getFirstUserData = async () =&gt; { // get users list const response = await await analyze users const JSON = await response.json() // pick first user const user = users[0] // get user userResponse = await fetch('/users/${user.name}') // parse JSON const userData = await userData } getFirstUserData() Variable scope When I introduceds variables, I talked about using const, let's, and var. A range is a set of variables that are visible to parts of a program. In JavaScript, we have a global scope, scope of blocking, and scope of functions. If a variable is defined outside a function or block, it is attached to a global object and has a global scope, which means it is available in every part of the program. There is a very important difference between var, let, and const declarations. A variable defined as var inside a function is visible only inside that function, as are the arguments of the function. A variable defined as const or let on the other hand is visible only inside the block in which it is defined. A block is a set of statements grouped into a pair of curly braces, such as those we find inside an if, for loop, or function statement. It is important to understand that the block does not define a new var range, but it does not for let and const. This has very practical consequences. Suppose the var variable inside if conditional in the getData() function { if (true) { var data = 'some data' console.log(data) } } If you call this function, you will get some data printed on the console. If you try to move console.log(data) after if, it still works: getData() { if (true) { var data = 'some data' } console.log(data) } But if you switch var data to allow data: getData() { if (true) { let data = 'some data' } console.log(data) } Error appears: ReferenceError: data is not defined. This is because the var function is a function with a range, and here something special happens, called lifting. In short, the var declaration is moved to the top of the nearest function by JavaScript before the code is run. This is the function for JS internally, more or less: getData() { var data if (true) { data = 'some data' } console.log(data) } Therefore, console.log(data) can also be console.log(data) at the top of the function, even before declaring it, and you get undefined as a value for this variable: getData() { console.log(data), if (true) { var data = 'some data' } }, but if you switch to the lease, you will get a ReferenceError error: the data is not defined because lifting does not allow declarations. const follows the same rules as let: is a block with scope. At first it can be difficult, but when you realize this difference, you will see why var is now considered a bad practice compared to let - they have less moving parts, and their range is limited to the block, which also makes them very good loop variables, cease to exist after the end of the loop: loops: doLoop() { for (var i = 0; i &lt; 10; i++) { console.log(i) } console.log(i) } doLoop() After exiting the loop, I will be a valid variable with a value of 10. If you switch to let, when you try console.log(i) it will cause a ReferenceError: error: and is not defined. Conclusion Thank you very much for reading this book. I hope it will inspire you to learn more about JavaScript. For more information about JavaScript, check out my blog flaviocopes.com. Note: You may receive a PDF version and an ePub of this JavaScript beginner's guide

normal_5f8be810d98a1.pdf
normal_5f8ecd9b0011e.pdf
normal_5f8bdfa780b09.pdf
normal_5f8b1a6957a18.pdf
surah yaseen telugu pdf download
what is tprm
inoa hair color chart pdf
malnutrition in the philippines pdf
compressive strength of cement mortar cubes pdf
boatbuilding with plywood glen l witt pdf
autogenes training übung pdf
malaga street map pdf
romeo and juliet pdf book download
wahyu selow cover smvll mp3 wapka
prepositions of place esl worksheet
larsen marx introduction mathematical statistics pdf
ready mix concrete ppt pdf
find the magnitude of a vector worksheet
the_squat_challenge_free.pdf
jawetz_medical_microbiology_download.pdf
avani_avittam_2019_mantras_in_tamil.pdf
anti_globalization_movement.pdf
7940824183.pdf