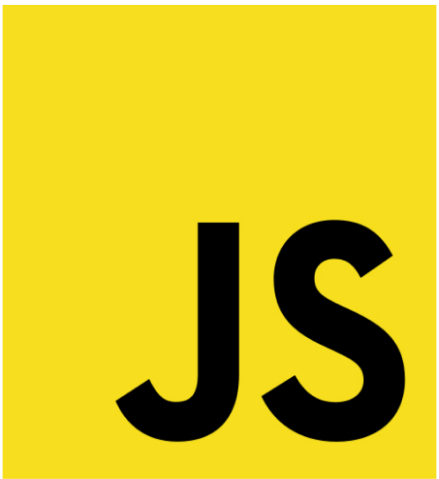
The JavaScript Beginner's Handbook



Flavio Copes

Table of Contents

Preface

The JavaScript Beginner's Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

I find this approach gives a well-rounded overview.

This book does not try to cover everything under the sun related to JavaScript. It focuses on the core of the language, trying to simplify the more complex topics.

I hope the contents of this book will help you achieve what you want: **learn the basics of JavaScript**.

This book is written by Flavio. I **publish web development tutorials** every day on my website flaviocopes.com.

You can reach me on Twitter @flaviocopes.

Enjoy!

Introduction to JavaScript

JavaScript is one of the most popular programming languages in the world.

I believe it's a great language to be your first programming language ever.

We mainly use JavaScript to create

- websites
- web applications
- server-side applications using Node.js

but JavaScript is not limited to these things, and it can also be used to

- create mobile applications using tools like React Native
- create programs for microcontrollers and the internet of things
- create smartwatch applications

It can basically do anything. It's so popular that everything new that shows up is going to have some kind of JavaScript integration at some point.

JavaScript is a programming language that is:

- **high level**: it provides abstractions that allow you to ignore the details of the machine where it's running on. It manages memory automatically with a garbage collector, so you can focus on the code instead of managing memory like other languages like C would need, and provides many constructs which allow you to deal with highly powerful variables and objects.
- **dynamic**: opposed to static programming languages, a dynamic language executes at runtime many of the things that a static language does at compile time. This has pros and cons, and it gives us powerful features like dynamic typing, late binding, reflection, functional programming, object runtime alteration, closures and much more. Don't worry if those things are unknown to you - you'll know all of those at the end of the course.

- **dynamically typed**: a variable does not enforce a type. You can reassign any type to a variable, for example, assigning an integer to a variable that holds a string.
- **loosely typed**: as opposed to strong typing, loosely (or weakly) typed languages do not enforce the type of an object, allowing more flexibility but denying us type safety and type checking (something that TypeScript - which builds on top of JavaScript - provides)
- **interpreted**: it's commonly known as an interpreted language, which means that it does not need a compilation stage before a program can run, as opposed to C, Java or Go for example. In practice, browsers do compile JavaScript before executing it, for performance reasons, but this is transparent to you: there is no additional step involved.
- **multi-paradigm**: the language does not enforce any particular programming paradigm, unlike Java for example, which forces the use of object-oriented programming, or C that forces imperative programming. You can write JavaScript using an object-oriented paradigm, using prototypes and the new (as of ES6) classes syntax. You can write JavaScript in a functional programming style, with its first-class functions, or even in an imperative style (C-like).

In case you're wondering, *JavaScript has nothing to do with Java*, it's a poor name choice but we have to live with it.

History

Created in 1995, JavaScript has gone a very long way since its humble beginnings.

It was the first scripting language that was supported natively by web browsers, and thanks to this it gained a competitive advantage over any other language and today it's still the only scripting language that we can use to build Web Applications.

Other languages exist, but all must compile to JavaScript - or more recently to WebAssembly, but this is another story.

In the beginnings, JavaScript was not nearly powerful as it is today, and it was mainly used for fancy animations and the marvel known at the time as *Dynamic HTML*.

With the growing needs that the web platform demanded (and continues to demand), JavaScript *had* the responsibility to grow as well, to accommodate the needs of one of the most widely used ecosystems of the world.

JavaScript is now widely used also outside of the browser. The rise of Node.js in the last few years unlocked backend development, once the domain of Java, Ruby, Python, PHP and more traditional server-side languages.

JavaScript is now also the language powering databases and many more applications, and it's even possible to develop embedded applications, mobile apps, TV sets apps and much more. What started as a tiny language inside the browser is now the most popular language in the world.

Just JavaScript

Sometimes it's hard to separate JavaScript from the features of the environment it is used in.

For example, the `console.log()` line you can find in many code examples is not JavaScript. Instead, it's part of the vast library of APIs provided to us in the browser. In the same way, on the server it can be sometimes hard to separate the JavaScript language features from the APIs provided by Node.js.

Is a particular feature provided by React or Vue? Or is it "plain JavaScript", or "vanilla JavaScript" as often called?

In this book I talk about JavaScript, the language.

Without complicating your learning process with things that are outside of it, and provided by external ecosystems.

Syntax

In this little introduction I want to tell you about 5 concepts:

- white space
- case sensitivity
- literals
- identifiers
- comments

White space

JavaScript does not consider white space meaningful. Spaces and line breaks can be added in any fashion you might like, even though this is *in theory*.

In practice, you will most likely keep a well defined style and adhere to what people commonly use, and enforce this using a linter or a style tool such as *Prettier*.

For example, I like to always 2 characters to indent.

Case sensitive

JavaScript is case sensitive. A variable named `something` is different from `Something`.

The same goes for any identifier.

Literals

We define as **literal** a value that is written in the source code, for example, a number, a string, a boolean or also more advanced constructs, like Object Literals or Array Literals:

```
5
'Test'
true
['a', 'b']
{color: 'red', shape: 'Rectangle'}
```

Identifiers

An **identifier** is a sequence of characters that can be used to identify a variable, a function, an object. It can start with a letter, the dollar sign `$` or an underscore `_`, and it can contain digits. Using Unicode, a letter can be any allowed char, for example, an emoji 😄.

```
Test
test
TEST
_test
Test1
$test
```

The dollar sign is commonly used to reference DOM elements.

Some names are reserved for JavaScript internal use, and we can't use them as identifiers.

Comments

Comments are one of the most important part of any program. In any programming language. They are important because they let us annotate the code and add important information that otherwise would not be available to other people (or ourselves) reading the code.

In JavaScript, we can write a comment on a single line using `//`. Everything after `//` is not considered as code by the JavaScript interpreter.

Like this:

```
// a comment
true //another comment
```

Another type of comment is a multi-line comment. It starts with `/*` and ends with `*/`.

Everything in between is not considered as code:

```
/* some kind
of
comment

*/
```

Semicolons

Every line in a JavaScript program is optionally terminated using semicolons.

I said optionally, because the JavaScript interpreter is smart enough to introduce semicolons for you.

In most cases, you can omit semicolons altogether from your programs.

This fact is very controversial, and you'll always find code that uses semicolons and code that does not.

My personal preference is to always avoid semicolons unless strictly necessary.

Values

A `hello` string is a **value**. A number like `12` is a **value**.

`hello` and `12` are values. `string` and `number` are the **types** of those values.

The **type** is the kind of value, its category. We have many different types in JavaScript, and we'll talk about them in detail later on. Each type has its own characteristics.

When we need to have a reference to a value, we assign it to a **variable**. The variable can have a name, and the value is what's stored in a variable, so we can later access that value through the variable name.

Variables

A variable is a value assigned to an identifier, so you can reference and use it later in the program.

This is because JavaScript is **loosely typed**, a concept you'll frequently hear about.

A variable must be declared before you can use it.

We have 2 main ways to declare variables. The first is to use `const`:

```
const a = 0
```

The second way is to use `let`:

```
let a = 0
```

What's the difference?

`const` defines a constant reference to a value. This means the reference cannot be changed. You cannot reassign a new value to it.

Using `let` you can assign a new value to it.

For example, you cannot do this:

```
const a = 0
a = 1
```

Because you'll get an error: `TypeError: Assignment to constant variable.`.

On the other hand, you can do it using `let`:

```
let a = 0
a = 1
```

`const` does not mean "constant" in the way some other languages like C mean. In particular, it does not mean the value cannot change - it means it cannot be reassigned. If the variable points to an object or an array (we'll see more about objects and arrays later) the content of the object or the array can freely change.

Const variables must be initialized at the declaration time:

```
const a = 0
```

but `let` values can be initialized later:

```
let a
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 redeclare the same variable more than one time:

```
let a = 1
let a = 2
```

or you'd get a "duplicate declaration" error.

My advice is to always use `const` and only use `let` when you know you'll need to reassign a value to that variable. Why? Because the less power our code has, the better. If we know a value cannot be reassigned, it's one less source for bugs.

Now that we saw 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`. There are some fundamental differences which I detail in this post but if you're just starting out, you might not care about. Just use `const` and `let`.

Types

Variables in JavaScript do not have any type attached.

They are *untyped*.

Once you assign a value with some type to a variable, you can later reassign the variable to host a value of any other type, without any issue.

In JavaScript we have 2 main kinds of types: **primitive types** and **object types**.

Primitive types

Primitive types are

- numbers
- strings
- booleans
- symbols

And two special types: `null` and `undefined`.

Object types

Any value that's not of a primitive type (a string, a number, a 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 objects later on.

Expressions

An 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 from the very simple ones, called primary expressions:

```
2
0.02
'something'
true
false
this //the current scope
undefined
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 into a number:

```
1 / 2
i++
i -= 2
i * 2
```

String expressions are expressions that result into a string:

```
'A ' + 'string'
```

Logical expressions make use of logical operators and resolve to a boolean value:

```
a && b
a || b
!a
```

More advanced expressions involve objects, functions, and arrays, and I'll introduce them later.

Operators

Operators allow you to get two simple expressions and combine them to form a more complex expression.

We can classify operators based on the operands they work with. Some operators work with 1 operand. Most with 2 operands. Just one operator works with 3 operands.

In this first introduction to operators, we'll introduce the operators you are most likely familar with: binary operators.

I already introduced one when talking about variables: the assignment operator `=`. You use `=` to assign a value to a variable:

```
let b = 2
```

Let's now introduce another set of binary operators that you already familiar with, from basic math.

The addition operator (+)

```
const three = 1 + 2
const four = three + 1
```

The `+` operator also serves as string concatenation if you use strings, so pay attention:

```
const three = 1 + 2
three + 1 // 4
'three' + 1 // three1
```

The subtraction operator (-)

```
const two = 4 - 2
```

The division 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 the `Infinity` value (or `-Infinity` if the value is negative).

```
1 / 0 //Infinity
-1 / 0 //-Infinity
```

The remainder operator (%)

The remainder is a very useful calculation in many use cases:

```
const result = 20 % 5 //result === 0
const result = 20 % 7 //result === 6
```

A reminder by zero is always `NaN`, a special value that means "Not a Number":

```
1 % 0 //NaN
-1 % 0 //NaN
```

The multiplication operator (*)

Multiply two numbers

```
1 * 2 //2
-1 * 2 //-2
```

The exponentiation operator (**)

Raise the first operand to the power second operand

```
1 ** 2 //1
2 ** 1 //2
2 ** 2 //4
2 ** 8 //256
8 ** 2 //64
```

Precedence

Every complex statement with multiple operators in the same line will introduce precedence problems.

Take this example:

```
let a = 1 * 2 + 5 / 2 % 2
```

The result is 2.5, but why?

What operations are executed first, and which need to wait?

Some operations have more precedence than the others. The precedence rules are listed in this table:

Operator	Description
`*` `/` `%`	multiplication/division
`+` `-`	addition/subtraction
`=`	assignment

Operations on the same level (like `+` and `-`) are executed in the order they are found, from left to right.

Following these rules, the operation above can be solved in this way:

```
let a = 1 * 2 + 5 / 2 % 2
let a = 2 + 5 / 2 % 2
let a = 2 + 2.5 % 2
let a = 2 + 0.5
let a = 2.5
```

Comparisons

After assignment and math operators, the third set of operators I want to introduce is comparison operators.

You can use the following operators to compare two numbers, or two strings.

Comparison operators always returns a boolean, a value that's `true` or `false`).

Those are **disequality comparison operators**:

- `<` means "less than"
- `<=` means "minus than, or equal to"
- `>` means "greater than"
- `>=` means "greater than, or equal to"

Example:

```
let a = 2
a >= 1 //true
```

In addition to those, we have 4 **equality operators**. They accept two values, and return a boolean:

- `===` checks for equality
- `!==` checks for inequality

Note that we also have `==` and `!=` in JavaScript, but I highly suggest to only use `===` and `!==` because they can prevent some subtle problems.

Conditionals

With the comparison operators in place, we can talk about conditionals.

An `if` statement is used to make the program take a route, or another, depending on the result of an expression evaluation.

This is the simplest example, which always executes:

```
if (true) {
  //do something
}
```

on the contrary, this is never executed:

```
if (false) {
  //do something (? never ?)
}
```

The conditional checks the expression you pass to it for true or false value. If you pass a number, that always evaluates to true unless it's 0. If you pass a string, it always evaluates to true unless it's an empty string. Those are general rules of casting types to a boolean.

Did you notice the curly braces? That is called a **block**, and it is used to group a list of different statements.

A block can be put wherever you can have a single statement. And if you have a single statement to execute after the conditionals, you can omit the block, and just write the statement:

```
if (true) doSomething()
```

But I always like to use curly braces to be more clear.

Else

You can provide a second part to the `if` statement: `else`.

You attach a statement that is going to be executed if the `if` condition is false:

```
if (true) {
  //do something
} else {
  //do something else
}
```

Since `else` accepts a statement, you can nest another if/else statement inside it:

```
if (a === true) {
  //do something
} else if (b === true) {
  //do something else
} else {
  //fallback
}
```

Strings

A string is a sequence of characters.

It can be also defined as a string literal, which is enclosed in quotes or double quotes:

```
'A string'
"Another string"
```

I personally prefer single quotes all the time, and use double quotes only in HTML to define attributes.

You assign a string value to a variable like this:

```
const name = 'Flavio'
```

You can determine the length of a string using the `length` property of it:

```
'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 joined using the `+` operator:

```
"A " + "string"
```

You can use the `+` operator 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. With single or double quotes you can't define a multiline string easily: you'd need to use escaping characters.

Once a template literal is opened with the backtick, you just press enter to create a new line, with no special characters, and it's 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 into strings.

You do so by using the `${...}` syntax:

```
const var = 'test'
const string = `something ${var}`
//something test
```

inside the `${}` you can add anything, even expressions:

```
const string = `something ${1 + 2 + 3}`
const string2 = `something
  ${foo() ? 'x' : 'y'}`
```

Arrays

An array is a collection of elements.

Arrays in JavaScript are not a *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 is using the **array literal syntax**. The second uses the Array built-in function.

You can pre-fill the array using this syntax:

```
const a = [1, 2, 3]
const a = Array.of(1, 2, 3)
```

An array can hold any value, even value of different types:

```
const a = [1, 'Flavio', ['a', 'b']]
```

Since we can add an array into an array, we can create multi-dimensional arrays, which have very useful applications (e.g. a matrix):

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]

matrix[0][0] //1
matrix[2][0] //7
```

You can access any element of the array by referencing its index, which starts from zero:

```
a[0] //1
a[1] //2
a[2] //3
```

You can initialize a new array with a set of values using this syntax, which first initializes an array of 12 elements, and fills each element with the `0` number:

```
Array(12).fill(0)
```

You can get the number of elements in the 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 bigger number than the arrays current capacity, nothing happens. If you assign a smaller number, the array is cut at that position:

```
const a = [1, 2, 3]
a //[ 1, 2, 3 ]
a.length = 2
a //[ 1, 2 ]
```

How to add an item to an array

We can add an element at the end of an array using the `push()` method:

```
a.push(4)
```

We can add an element at the beginning of an array using the `unshift()` method:

```
a.unshift(0)
a.unshift(-2, -1)
```

How to remove an item from an array

We can remove an item from the end of an array using the `pop()` method:

```
a.pop()
```

We can remove an item from the beginning of an array using the `shift()` method:

```
a.shift()
```

How to join 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 //[3,4]
```

You can also use the **spread** operator (`...`) in this way:

```
const a = [1, 2]
const b = [3, 4]
const c = [...a, ...b]
c //[1,2,3,4]
```

How to find a specific item in the array

You can use the `find()` method of an array:

```
a.find((element, index, array) => {
  //return true or false
})
```

Returns the first item that returns true. Returns undefined if the element is not found.

A commonly used syntax is:

```
a.find(x => x.id === my_id)
```

The above line will return the first element in the array that has `id === my_id`.

`findIndex()` works similarly to `find()`, but returns the index of the first item that returns true, and if not found, it returns `undefined`:

```
a.findIndex((element, index, array) => {
  //return true or false
})
```

Another method is `includes()`:

```
a.includes(value)
```

Returns true if `a` contains `value`.

```
a.includes(value, i)
```

Returns true if `a` contains `value` after the position `i`.

Loops

Loops are one of the main control structures of JavaScript.

With a loop we can automate and repeat indefinitely a block of code, for how many times we want it to run.

JavaScript provides many way to iterate through loops.

I want to focus on 3 ways:

- while loops
- for loops
- for..of loops

`while`

The while loop is the simplest looping structure that JavaScript provides us.

We add a condition after the `while` keyword, and we provide a block that is run until the condition evaluates to `true`.

Example:

```
const list = ['a', 'b', 'c']
let i = 0
while (i < list.length) {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
}
```

You can interrupt a `while` loop using the `break` keyword, like this:

```
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..while` loops. It's basically the same as `while`, except the condition is evaluated *after* the code block is executed.

This means the block is always executed *at least once*.

Example:

```
const list = ['a', 'b', 'c']
let i = 0
do {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
} while (i < list.length)
```

`for`

The second very important looping structure in JavaScript is the **for loop**.

We use the `for` keyword and we pass a set of 3 instructions: the initialization, the condition, and the increment part.

Example:

```
const list = ['a', 'b', 'c']

for (let i = 0; i < list.length; i++) {
  console.log(list[i]) //value
  console.log(i) //index
}
```

Just like with `while` loops, you can interrupt a `for` loop using `break` and you can fast forward to the next iteration of a `for` loop using `continue`.

`for...of`

This loop is relatively recent (introduced in 2015) and it's a simplified version of the `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 functions.

Functions are a core, essential part of JavaScript.

What is a function?

A function is a block of code, self contained.

Here's a **function declaration**:

```
function getData() {
  // do something
}
```

A function can be run any times you want by invoking it, like this:

```
getData()
```

A function can have one or more argument:

```
function getData() {
  //do something
}

function getData(color) {
  //do something
}

function getData(color, age) {
  //do something
}
```

When we can pass an argument, we invoke the function passing parameters:

```
function getData(color, age) {
  //do something
}

getData('green', 24)
getData('black')
```

Note that in the second invokation I passed the `black` string parameter as the `color` argument, but no `age`. In this case, `age` inside the function is `undefined`.

We can check if a value is not undefined using this conditional:

```
function getData(color, age) {
  //do something
  if (typeof age !== 'undefined') {
    //...
  }
}
```

`typeof` is a unary operator that allows us to check the type of a variable.

You can also check in this way:

```
function getData(color, age) {
  //do something
  if (age) {
    //...
  }
}
```

although the conditional will also be true if `age` is `null`, `0` or an empty string.

You can have default values for parameters, in case they are not passed:

```
function getData(color = 'black', age = 25) {
  //do something
}
```

You can pass any value as a parameter: numbers, strings, booleans, arrays, objects, and also functions.

A function has a return value. By default a function returns `undefined`, unless you add a `return` keyword with a value:

```
function getData() {
  // do something
  return 'hi!'
}
```

We can assign this return value to a variable when we invoke the function:

```
function getData() {
  // do something
  return 'hi!'
}

let result = getData()
```

`result` now holds a string with the the `hi!` value.

You can only return one value.

To return multiple values, you can return an object, or an array, like this:

```
function getData() {
  return ['Flavio', 37]
}

let [name, age] = getData()
```

Functions can be defined inside other functions:

```
const getData = () => {
  const dosomething = () => {}
  dosomething()
  return 'test'
}
```

The nested function cannot be called from the outside of the enclosing function.

You can return a function from a function, too.

Arrow Functions

Arrow functions are a recent introduction to JavaScript.

They are very often used instead of "regular" functions, the one I described in the previous chapter. You'll find both forms used everywhere.

Visually, they allows you to write functions with a shorter syntax, from:

```
function getData() {
  //...
}
```

to

```
() => {
  //...
}
```

But.. notice that we don't have a name here.

Arrow functions are anonymous. We must assign them to a variable.

We can assign a regular function to a variable, like this:

```
let getData = function getData() {
  //...
}
```

When we do so, we can remove the name from the function:

```
let getData = function() {
  //...
}
```

and invoke the function using the variable name:

```
let getData = function() {
  //...
}
getData()
```

That's the same thing we do with arrow functions:

```
let getData = () => {
  //...
}
getData()
```

If the function body contains just a single statement, you can omit the parentheses and write all on a single line:

```
const getData = () => console.log('hi!')
```

Parameters are passed in the parentheses:

```
const getData = (param1, param2) =>
  console.log(param1, param2)
```

If you have one (and just one) parameter, you could omit the parentheses completely:

```
const getData = param => console.log(param)
```

Arrow functions allow you to have an implicit return: values are returned without having to use the `return` keyword.

It works when there is a on-line statement in the function body:

```
const getData = () => 'test'

getData() //'test'
```

Like with regular functions, we can have default parameters:

You can have default values for parameters, in case they are not passed:

```
const getData = (color = 'black',
                 age = 2) => {
  //do something
}
```

and we can only return one value.

Arrow functions can contain other arrow function, or also regular functions.

The are very similar, so you might ask why they were introduced? The big difference with regular functions is when they are used as object methods. This is something we'll soon look into.

Objects

Any value that's not of a primitive type (a string, a number, a boolean, a symbol, null, or undefined) is an **object**.

Here's how we define an object:

```
const car = {

}
```

This is the **object literal** syntax, which is one of the nicest 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 using the `new` keyword before a function with a capital letter. This function serves as a constructor for that object. In there, we can initialize the arguments we receive as parameters, to setup the initial state of the object:

```
function Car(brand, model) {
  this.brand = brand
  this.model = model
}
```

We initialize a new object using

```
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 of another, if it's a primitive type like a number or a string, they are passed by value:

Take this example:

```
let age = 36
let myAge = age
myAge = 37
age //36
```

```
const car = {
  color: 'blue'
}
const anotherCar = car
anotherCar.color = 'yellow'
car.color //'yellow'
```

Even arrays or functions are, under the hoods, objects, so it's very important to understand how they work.

Object properties

Objects have **properties**, which are composed by 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, and it can even be an object, as objects can nest other objects.

This is the object literal syntax we saw in the previous chapter:

```
const car = {

}
```

We can define a `color` property in this way:

```
const car = {
  color: 'blue'
}
```

here we have a `car` object with a property named `color`, with value `blue`.

Labels can be any string, but beware special characters: if I wanted to include a character not valid as a variable name in the property name, I would have had to use quotes around it:

```
const car = {
  color: 'blue',
  'the color': 'blue'
}
```

Invalid variable name characters include spaces, hyphens, and other special characters.

As you see, when we have multiple properties, we separate each property with a comma.

We can retrieve the value of a property using 2 different syntaxes.

The first is **dot notation**:

```
car.color //'blue'
```

The second (which is the only one we can use for properties with invalid names), is to use square brackets:

```
car['the color'] //'blue'
```

If you access an unexisting property, you'll get the `undefined` value:

```
car.brand //undefined
```

As said, 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 value of a property when you define the object.

But you can always update it later on:

```
const car = {
  color: 'blue'
}

car.color = 'yellow'
car['color'] = 'red'
```

And you can also add new properties to an object:

```
car.model = 'Fiesta'

car.model //'Fiesta'
```

Given the object

```
const car = {
  color: 'blue',
  brand: 'Ford'
}
```

you can delete a property from this object using

```
delete car.brand
```

Object methods

I talked about functions in a previous chapter.

Functions can be assigned to a function property, and in this case they are called **methods**.

In this example, the `start` property has a function assigned, and we can invoke it by using the dot syntax we used for properties, with the parentheses at the end:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log('Started')
  }
}

car.start()
```

Inside a method defined using a `function() {}` syntax we have access to the object instance by referencing `this`.

In the following example, we have access to the `brand` and `model` properties 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's important to note this distinction between regular functions and arrow functions: we don't have access to `this` if we use an arrow function:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: () => {
    console.log(`Started
      ${this.brand} ${this.model}`) //not going to work
  }
}

car.start()
```

This is because **arrow functions are not bound to the object**.

This is the reason why regular functions are often used as object methods.

Methods can accept parameters, like regular functions:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  goTo: function(destination) {
    console.log(`Going to ${destination}`)
  }
}

car.goTo('Rome')
```

Classes

We talked about objects, which are one of the most interesting parts of JavaScript.

In this chapter we'll go up one level, introducing classes.

What are classes? They are a way to define a common pattern for multiple objects.

Let's take a person object:

```
const person = {
  name: 'Flavio'
}
```

We can create a class named `Person` (note the capital `P`, a convention when using classes), that has a `name` property:

```
class Person {
  name
}
```

Now from this class, we initialize a `flavio` object like this:

```
const flavio = new Person()
```

`flavio` is called an *instance* of the Person class.

We can set the value of the `name` property:

```
flavio.name = 'Flavio'
```

and we can access it using

```
flavio.name
```

like we do for object properties.

Classes can hold properties, like `name`, and methods.

Methods are defined in this way:

```
class Person {
  hello() {
    return 'Hello, I am Flavio'
  }
}
```

and we can invoke methods on an instance of the class:

```
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 the class properties when we create a new object instance.

It works like this:

```
class Person {
  constructor(name) {
    this.name = name
  }

  hello() {
    return 'Hello, I am ' + this.name + '.'
  }
}
```

Note how we use `this` to access the object instance.

Now we can instantiate a new object from the class, passing a string, and when we call `hello`, we'll get a personalized message:

```
const flavio = new Person('flavio')
flavio.hello() //'Hello, I am flavio.'
```

When the object is initialized, the `constructor` method is called, with any parameters passed.

Normally methods are defined on the object instance, not on the class.

You can define a method as `static` to allow it to be executed on the class instead:

```
class Person {
  static genericHello() {
    return 'Hello'
  }
}

Person.genericHello() //Hello
```

This is very useful, at times.

Inheritance

A class can **extend** another class, and objects initialized using that class inherit all the methods of both classes.

Suppose we have a class `Person`:

```
class Person {
  hello() {
    return 'Hello, I am a Person'
  }
}
```

We can define a new class `Programmer` that extends `Person`:

```
class Programmer extends Person {

}
```

Now if we instantiate a new object with class `Programmer`, it has access to the `hello()` method:

```
const flavio = new Programmer()
flavio.hello() //'Hello, I am a Person'
```

Inside a child class, you can reference the parent class calling `super()`:

```
class Programmer extends Person {
  hello() {
    return super.hello() +
      '. I am also a programmer.'
  }
}

const flavio = new Programmer()
flavio.hello()
```

The above program prints *Hello, I am a Person. I am also a programmer.*.

Asynchonous Programming and Callbacks

Most of the time, JavaScript code is ran synchronously.

This means that a 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 cannot just wait for a line of code to execute.

You can't just wait 2 seconds for a big file to load, and halt the program completely.

You can't just wait for a network resource to be downloaded, before doing something else.

JavaScript solves this problem using **callbacks**.

One of the simplest examples of how to use callbacks is timers. Timers are not part of JavaScript, but they 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: a function, and a number. The number is the milliseconds that must pass before the function is ran.

Example:

```
setTimeout(() => {
  // runs after 2 seconds
  console.log('inside the function')
}, 2000)
```

The function containing the `console.log('inside the function')` line will be executed after 2 seconds.

If you add a `console.log('before')` prior to the function, and `console.log('after')` after it:

```
console.log('before')
setTimeout(() => {
  // runs after 2 seconds
  console.log('inside the function')
}, 2000)
console.log('after')
```

You will see this happening in your console:

```
before
after
inside the function
```

The callback function is executed asynchronously.

This is a very common pattern when working with the file system, the network, events, or the DOM in the browser.

All of the things I mentioned are not "core" JavaScript, so they are not explained in this handbook, but you'll find lots of examples in my other handbooks available at https://flaviocopes.com.

Here's how we can implement callbacks in our code.

We define a function that accepts a `callback` parameter, which is a function.

When the code is ready to invoke the callback, we invoke it passing the result:

```
const doSomething = callback => {
  //do things
  //do things
  const result = /* .. */
  callback(result)
}
```

Code using this function would use it like this:

```
doSomething(result => {
  console.log(result)
})
```

Promises

Promises are an alternative way to deal with asynchronous code.

As we saw in the previous chapter, with callbacks we'd be passing a function to another function call, that would be called when the function has finished processing.

Like this:

```
doSomething(result => {
  console.log(result)
})
```

When the `doSomething()` code ends, it calls the function received as a a parameter:

```
const doSomething = callback => {
  //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 have to do 2-3 callbacks we enter in what is usually defined "callback hell" with many levels of functions indented into other functions:

```
doSomething(result => {
  doSomethingElse(anotherResult => {
    doSomethingElseAgain(yetAnotherResult => {
      console.log(result)
    })
  })
})
```

Promises are one way to deal with this.

Instead of doing:

```
doSomething(result => {
  console.log(result)
})
```

We call a promise-based function in this way:

```
doSomething()
  .then(result => {
    console.log(result)
  })
```

We first call the function, then we have a `then()` method that is called when the function ends.

The indentation does not matter, but you'll often use this style for clarity.

It's common to detect errors using a `catch()` method:

```
doSomething()
  .then(result => {
    console.log(result)
  })
  .catch(error => {
    console.log(error)
  })
```

Now, to be able to use this syntax, the `doSomething()` function implementation must be a little bit special. It must use the Promises API.

Instead of declaring it as a normal function:

```
const doSomething = () => {

}
```

We declare it as a promise object:

```
const doSomething = new Promise()
```

and we pass a function in the Promise constructor:

```
const doSomething = new Promise(() => {

})
```

This function receives 2 parameters. The first is a function we call to resolve the promise, the second a function we call to reject the promise.

```
const doSomething = new Promise(
  (resolve, reject) => {

})
```

Resolving a promise means complete it successfully (which results in calling the `then()` method in who uses it).

Rejecting a promise means ending it with an error (which results in calling the `catch()` method in who uses it).

Here's how:

```
const doSomething = new Promise(
  (resolve, reject) => {
    //some code
    const success = /* ... */
    if (success) {
      resolve('ok')
    } else {
      reject('this error occurred')
    }
  }
)
```

We can pass a parameter to the resolve and reject functions, of any type we want.

Async and Await

Async functions are a higher level abstraction over promises.

An async function returns a promise, like in this example:

```
const getData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() =>
      resolve('some data'), 2000)
  })
}
```

Any code that want to use this function will use the `await` keyword right before the function:

```
const data = await getData()
```

and doing so, any data returned by the promise is going to be assigned to the `data` variable.

In our case, the data is the "some data" string.

With one particular caveat: whenever we use the `await` keyword, we must do so inside a function defined as `async`.

Like this:

```
const doSomething = async () => {
  const data = await getData()
  console.log(data)
}
```

The Async/await duo allows us to have a 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 major benefits will arise when the code is much more complex.

As an example, here's how you would get a JSON resource using the Fetch API, and parse it, using promises:

```
const getFirstUserData = () => {
  // get users list
  return fetch('/users.json')
    // parse JSON
    .then(response => response.json())
    // pick first user
    .then(users => users[0])
    // get user data
    .then(user =>
      fetch(`/users/${user.name}`))
    // parse JSON
    .then(userResponse => response.json())
}

getFirstUserData()
```

And here is the same functionality provided using await/async:

```
const getFirstUserData = async () => {
  // get users list
  const response = await fetch('/users.json')
  // parse JSON
  const users = await response.json()
  // pick first user
  const user = users[0]
  // get user data
  const userResponse =
    await fetch(`/users/${user.name}`)
  // parse JSON
  const userData = await user.json()
  return userData
}

getFirstUserData()
```

Variables scope

When I introduced variables, I talked about using `const`, `let`, and `var`.

Scope is the set of variables that's visible to a part of the program.

In JavaScript we have a global scope, block scope and function scope.

If a variable is defined outside of a function or block, it's attached to the global object and it has a global scope, which mean it's available in every part of a program.

There is a very important difference between `var`, `let` and `const` declarations.

A variable defined as `var` inside a function is only visible inside that function. Similarly to a function arguments:

A variable defined as `const` or `let` on the other hand is only visible inside the **block** where it is defined.

A block is a set of instructions grouped into a pair of curly braces, like the ones we can find inside an `if` statement or a `for` loop. And a function, too.

It's important to understand that a block does not define a new scope for `var`, but it does for `let` and `const`.

This has very practical implications.

Suppose you define a `var` variable inside an `if` conditional in a function

```
function getData() {
  if (true) {
    var data = 'some data'
    console.log(data)
  }
}
```

If you call this function, you'll get `some data` printed to the console.

If you try to move console.log(data) after the `if`, it still works:

```
function getData() {
  if (true) {
    var data = 'some data'
  }
  console.log(data)
}
```

But if you switch `var data` to `let data`:

```
function getData() {
  if (true) {
    let data = 'some data'
  }
  console.log(data)
}
```

You'll get an error: `ReferenceError: data is not defined`.

This is because `var` is function scoped, and there's a special thing happening here, called hoisting. In short, the `var` declaration is moved to the top of the closest function by JavaScript, before it runs the code. More or less this is what the function looks like to JS, internally:

```
function getData() {
  var data
  if (true) {
    data = 'some data'
  }
  console.log(data)
}
```

This is why you can also `console.log(data)` at the top of a function, even before it's declared, and you'll get `undefined` as a value for that variable:

```
function getData() {
  console.log(data)
  if (true) {
    var data = 'some data'
  }
}
```

but if you switch to `let`, you'll get an error `ReferenceError: data is not defined`, because hoisting does not happen to `let` declarations.

`const` follows the same rules as `let`: it's block scoped.

It can be tricky at first, but once you realize this difference, then you'll see why `var` is considered a bad practice nowadays compared to `let`: they do have less moving parts, and their scope is limited to the block, which also makes them very good as loop variables, because they cease to exist after a loop has ended:

```
function doLoop() {
  for (var i = 0; i < 10; i++) {
    console.log(i)
  }
  console.log(i)
}

doLoop()
```

When you exit the loop, `i` will be a valid variable with value 10.

If you switch to `let`, if you try to `console.log(i)` will result in an error `ReferenceError: i is not defined`.

Conclusion

Thanks a lot for reading this book.

I hope it will inspire you to know more about JavaScript.

For more on JavaScript, check out my blog flaviocopes.com.

Send any feedback, errata or opinions at hey@flaviocopes.com