sofia

LAST UPDATED

Febuary 2023

WRITTEN BY

Sofia Franek

Understanding JavaScript

Written by Sofia Franek

WEBSITE

www.sofiafranek.com

LINKEDIN

sofia-franek

MEDIUM

@sofiafranek_

Index

1.0 Syntax Parser

By default, your computer can't read JavaScript code by itself. When you run JavaScript code, a compiler will convert your code into instructions that your machine can understand.

We can think of the syntax parser as an interpreter between your code and your computer. It goes through your code and determines if the syntax is valid JavaScript before executing the code. JavaScript is fault-tolerant, so it works intelligently and tries to guess best and understand what you want.

Overall, our code isn't directly executed by the computer. It's passed via a JavaScript engine that sets up the execution context and interprets the code.

1.1 Creation Phase

In JavaScript, the syntax parser wraps the executable code in an execution context. This execution context is a concept of the environment our code is running on, either a global environment or a function environment.
The global execution context is accessible everywhere in our code. This context creates two things for us: the global object and the variable called this.

'this' keyword

The keyword this refers to an object that is executing a piece of code. It always refers to an object that is executing the current function.

If the function being referenced is a regular function it means that this is referencing the global object.

If the function being referenced is a method in an object, the this keyword is referencing the object itself.

1.2 Creation and Hoisting

JavaScript is a single-threaded language which means it can run one command at a time.

Creation

When code is executed in a JavaScript application it goes through two phases; creation and execution phase.

Creation phase

When the application goes through the creation phase, the compiler runs through the entire code. It stores in its memory all the function and variable declarations. All variables are given a value of undefined. If there are any conflicts between functions and variable declarations the variable is forgotten and the function is remembered.

Execution phase

During the execution phase, the variables are assigned values and the functions are executed.

Hoisting

If we want to invoke a function before its creation, you would think you would get an error. But not with JavaScript.

```
myFunction();
console.log(myVar);

let myVar = "I'm a variable";
const myFunction = () => {
  console.log("I'm a function"
}

/* Result -->
"I'm a variable"
undefined
*/
```

In this example, JavaScript runs the function but shows the variable to be undefined. This is because, in JavaScript, variables and functions are available, even if they are written later in your code. This happens because, during the creation phase of the code, memory is set up for variables and functions — this is called hoisting.

Before code gets executed in JavaScript, the JavaScript engine already has all the variables and functions in its memory, which is why you can call them. However, variables by default have a value of undefined until you define a value for them yourself in your code. So in the example above, that is why we received undefined back, so make sure you define your variables before you use them!

1.3 Strict mode

In JavaScript, the strict mode was introduced as a feature to allow us to place a program or function in a strict operating context. This strict context stops certain actions from being taken. The 'use strict' statement tells the browser to use the strict mode.

Strict mode can be used in two ways in JavaScript. In a global scope for the entire script or applied to an individual function.

Using strict mode for the entire script

The syntax used to enable strict mode is using the statement 'use strict' at the top of our script file.

```
// Script file strict mode syntax
'use strict';
```

It's recommended to enable strict mode on functions as you can get into a situation where you're concatenating strict and non-strict scripts together.

Using strict mode for a function

To enable strict mode for a function we put the same statement for entire scripts, 'use strict'but instead of at the top of the file, we put it in the function's body before any other statements.

```
const myFunction = () => {
  // Function level strict mode syntax
  'use strict';

  return "strict mode function!";
};
```

Benefits of using ’use strict'

- Eliminates JavaScript silent errors by changing them to throw errors.
- Fixes mistakes that make it difficult for JavaScript to perform optimisations.
- Disables features that don’t make sense or are poorly executed.
- Stops some syntax likely to be defined in future versions of ECMAScript.
- Makes it easier to write secure JavaScript.

1.4 Function statement vs. function expression

Function statement

A function statement declares a function, which is saved and expected later when it's called in the code. Function statements must begin using the function keyword and are invoked by using the declared function name.

A function statement gets hoisted during the creation phase of the execution context, so the statement is available in your application memory.

```
function hello() {
  console.log("Hello world!");
}
```

Function expression

An expression is a unit of code that results in a value.
Function expressions load only when the applications interpreter reaches the line of code the function lives on. They can’t be hoisted which allows them to keep a copy of the local variables from the scope they were defined in.

A function expression can be stored in a variable, this means the variable can be used as a function. Functions that are stored in variables do not need function names. They are invoked using the variable name.

```
let hello = function() {
  console.log("Hello world!");
}
```

1.5 Function programming

In JavaScript, we can enforce functional programming. In general, implementing functional programming leads to less code, as JavaScript already has a lot of built-in functions for common uses. We can architect our application using pure and isolated functions and avoid mutability and side effects.

Pure functions

A pure function in JavaScript is one that when you reapply the results to that function again, won't produce a different result. The same input will always give the same output and will have no side effects.

Isolated functions

When using isolated functions, you have no dependence on the state of your application. Anything you need for your function to be carried out you pass into the function as an argument.

Side effects

Side effects are when your code interacts with anything outside the function that would change the data in that function.

Mutability

Don't change things in your functions once you've made them. If you need to change something in your data, make changes to a copy. When you start to change your functions further down in your code, you can run into mutability and side effects. This is not clean coping and isn't advised when functional programming.

Higher-order functions

Higher-order functions are functions that take other functions as their argument. The most common higher-order functions in ES6 are map , filter and reduce .

map()

The map method loops through each piece of data in an array instead of having to use a loop.

```
const arr1 = [1, 2, 3];
const arr2 = arr1.map(value => value * 2);

// Result --> [2, 4, 6]
console.log(arr2)
```

I talk about filter() and reduce() in my article Common Methods in JavaScript found in my other ebook JavaScript Fundamentals.

1.6 JSON and object literals

JSON and object literals are very similar in syntax. The only noticeable difference between the JSON and object literal syntax is that in JSON both keys and values are wrapped in double quotes, whereas in object literals it's just the values. JavaScript also has a built-in conversion between JSON and JavaScript objects. If you want to convert an object to JSON data, you can use JSON.stringify(object) .

Object literal

An object literal is when you declare an object in JavaScript which is a variable that stores data in key-value pairs. The object literal syntax is the simplest way to create JavaScript objects.

```
const data = {
  firstName: "Sofia",
  lastName: "Franek"
}
```

JSON

JSON is a widely used language for storing and transferring data that is based on JavaScript objects. It copies syntax from the JavaScript object literal which is why JSON and object literals are very similar looking.

JSON data is stored as a separate file from JavaScript, while object literals appear within your JavaScript code files.

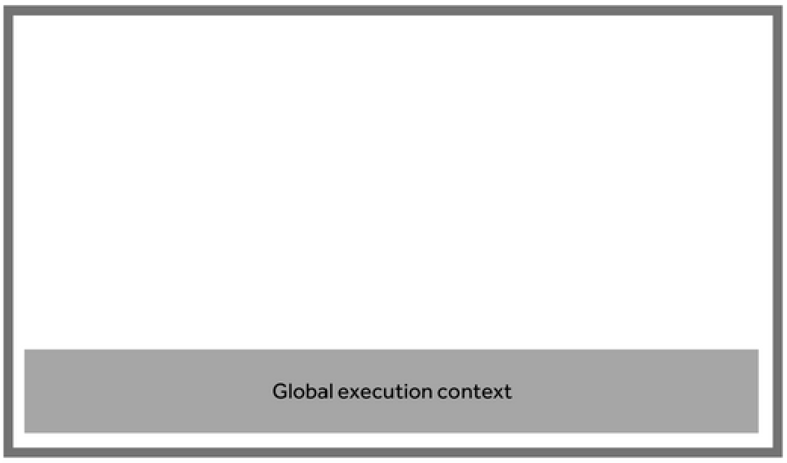
```
{
  "firstName": "Sofia",
  "lastName": "Franek"
}
```

1.7 Execution stack

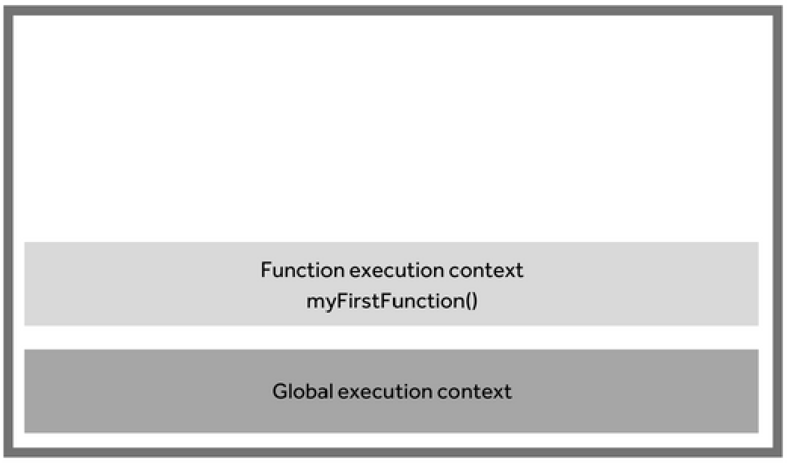
The execution stack in JavaScript keeps track of all the execution contexts created during the life cycle of a script. Because JavaScript is a single-threaded language, this means it's only capable of executing a single task at a time. So when actions, functions and events occur in your code it creates an execution context for each one. This results in a stack of piled-up execution contexts to be executed known as the execution stack.

How it works

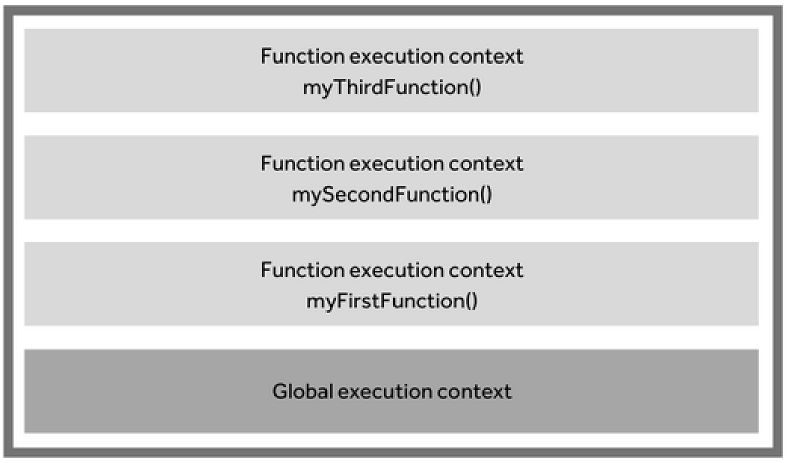
The script is first loaded into the JavaScript engine, after this, the JavaScript engine will create the global execution content and places it at the base of the execution stack. The global execution content is only for JavaScript code that is not inside of a function.



When the JavaScript engine encounters its first function, a new function execution context is created and added to the execution stack. This new context is placed on top of the current context.



During the first function call, its execution context becomes the active context. When a second function gets called it gets placed on the top of the stack and becomes the active context instead. Once functions get executed completely, they are popped out of the stack.


Function execution context
myThirdFunction()
Function execution context
mySecondFunction()
Function execution context
myFirstFunction()
Global execution context

When the execution of all the code is completed, the JavaScript engine removes the global execution context from the current stack.

1.8 bind(), call() and apply()

In JavaScript, everything is an object, and because everything is an object we have access to properties to functions. The keyword this is a property given to every function automatically and is used inside a function allowing us to refer to an object that invokes the function where the this keyword is used.

.bind()

The bind()method in JavaScript creates a new function that when called has its own this keyword set to the provided value. This allows us to define the value of this when calling a function.

```
let person = {
  data: [
    {name: "Sofia"}
  ];

  clickHandler() {
    console.log(this.data[0].name)
  }
};

// Result --> Error (this = button)
// There is no 'this' on the button element
$("button").click(person.clickHandler);

// Result --> Sofia (this = person object)
$("button").click(person.clickHandler.bing(person);)
```

.call()

The call() method calls a function with a this value and arguments. So we can call any function and state what this should reference within the calling function. This method doesn't make a copy of the function it is being called on.

```
// functionName.call(object, functionArguments)

let obj = { number: 5 }
let addFunc = function(a, b) {
  return this.number + a + b;
}

// Result --> 8
console.log(addFunc.call(obj, 1, 2));
```

.apply()

The call()and apply()methods serve the exact same purpose. The difference between them is that call()expects all parameters to be passed in individually, whereas apply()expects an array of all parameters.

```
let obj = { number: 5 }
let addFunc = function(a) {
  return this.number + a;
}

let array = [1, 2]

// Result --> 8
console.log(addFunc.call(obj, array));
```

1.9 Closures and Callbacks

Closures

A closure in JavaScript are functions that are nested in other functions. Mostly used to avoid scope clash with other parts of your JavaScript application.

In the chapter, Execution Stack 1.7 we know that after a function is returned, it is removed from the execution stack, but the function nested inside the function can still access the parent function variables, and this is known as closures.

```
let myVar = 1;

function outerFunction() {
  let mySecondVar = 5;

  function innerFunction() {
    let myThirdVar = 10;
    return myVar + mySecondVar + myThirdVar;
  }

  return innerFunction;
}

// Result --> 16
outerFunction();
```

Callbacks

A callback in JavaScript is a function that is passed into another function as an argument to be executed later. Functions can also be returned as the result of another function. When functions are passed in as arguments they aren't executed immediately, they are called when the function that you passed the callback function into calls it.

```
let arr = [1, 2, 3, 4, 5];

function checkIsOddNum(num) {
  return num % 2 !== 0;
}

function allOddNums = arr.filter(checkIsOddNum);

// Result --> [1, 3, 5]
console.log(allOddNums)
```

2.0 Variable Scope

The variable scope in JavaScript is where variables live. If a variable is defined outside of a function, it is accessible anywhere in the code.

Variables that are local live inside a function and will have a different scope for every call of that function.

Variables that are defined with the keywords let and const are not accessible outside of the block of code they were defined in.

If a variable is defined with the const keyword it cannot be reassigned.

```
let myVar = 10;

if (myVar === 10) {
  let myVar = 20;
  console.log(myVar, "In function");
}

console.log(myVar, "Outside of function");

/*
  Result -->
  20
  10
*/
```

2.1 The Scope Chain

In JavaScript, scope is a way of accessing variables, objects and functions in our code. With scope, we can access what we need. There are two kinds of scope: Global scope and Local scope. I talk more about these in detail in this article- Variables in JavaScript.

The scope chain establishes the scope for a given function. So each time a function is defined they have its own nested scope and any function defined within another function has a local scope which is linked to its outer function. This is what's known as the chain.

```
const myFunction = () => {
  console.log(myVar)
}

let myVar = 10;
// Result --> 10
myFunction();
```

2.2 Values vs. references

In JavaScript, we can pass by value and by reference. Passing by value happens when assigning primitives while passing by reference happens when assigning objects.

Values

In JavaScript, when we are passing by value all primitive values are passed by value. Every time we assign a value to a variable, a copy of that value is created. This is what's known as passing by value. This happens every single time.

```
let myFirstVar = 10;
let mySecondVar = myFirstVar;

mySecondVar = mySecondVar + 5;

// Result --> 10
console.log(myFirstVar);

// Result --> 15
console.log(mySecondVar);
```

In this example, we can see myFirstVar defined a variable initialized with the number 10. The next variable mySecondVar is initialized with the value of myFirstVar — which is passing-by value. A copy of myFirstVar is assigned to mySecondVar. When we increase mySecondVar by 5 we change the mySecondVar variables value but it doesn't affect myFirstVar the initial variable value.

References

When creating objects, we're given a reference to that object. If two variables hold the same reference, then when we change the object it reflects in both variables.

```
let myFirstVar = [10];
let mySecondVar = myFirstVar;

mySecondVar.push(5)

// Result --> [1, 2]
console.log(myFirstVar);

// Result --> [1, 2]
console.log(mySecondVar);
```

In this example, we can see myFirstVar creates an array and defines a variable with the reference to the created array. The next variable mySecondVar initializes with mySecondVar with the reference stored myFirstVar — this is a pass-by-reference. mySecondVar.push(5) mutates the array by pushing the number 2, because myFirstVar and mySecondVar both reference the same array, this change affects both variables.

2.3 Spread operator

In JavaScript, the spread operator allows us to expand or spread an iterable or an array. We can use the spread operator in a number of ways in JavaScript.

Using the spread operator with an object

We can use the spread operator with object literals.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3 };

// joining obj1 and obj2 to obj3 into a variable
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Clone an array using the spread operator

```
let arr1 = [1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// add a new value to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

In the above example, you can see how variables arr1 and arr2 are both referencing the same array. This is why a change in one variable changes the result in both. However, if we want to copy the arrays but not keep them linked to each we can use the spread operator. This means the change in one array, doesn't change the other.

```
let arr1 = [1, 2, 3];

// Copy arr1 using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// add a new value to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Copy an array using the spread operator

We can use the spread operator to copy the items into a single array.

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4, 5];

console.log(arr2);
/* Result -->
  [1, 2, 3, 4, 5]
*/
```

2.4 Inheritance and the Prototype Chain

Inheritance: An object getting access to the property and the methods of another object.

In JavaScript, we have one construct for inheritance- an object. Each object has its own private property that holds a link to another object called its prototype. So this goes on, that prototype object has a prototype of its own, and this continues until we reach null which has no prototype and acts as the final link in the chain.

Inheritance with the prototype chain

The prototype is a reference to another object.

JavaScript objects have a link to a prototype object. When we try to access a property of an object, the property will be searched not only on the object but on all the objects in the chain until a matching name is found or the end of the prototype chain is reached.

```
// { a: 1, b: 2 } -> { b: 3, c: 4 } -> { d: 5 } -> Object.prototype -> null
// __proto__ sets the [[Prototype]]

const myObject= {
  a: 1,
  b: 2,
  __proto__: {
    b: 3,
    c: 4,
    __proto__: {
      d: 5,
    },
  },
};

// Result --> 5
console.log(myObject); // 5
```

2.5 Prototype

Every function in JavaScript has a prototype, but only when a function is a constructor will the prototype be used. The prototype, however, is not the function's prototype it's the prototype of every object that's being created if you're using it as a constructor.

We can use the new keyword to set the prototype to the function's prototype.

```
function FullName(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let person = new FullName("Sofia", "Franek");
FullName.prototype.getFullName = () => this.firstName + this.lastName

person.getFullName();
```

We can also add features to objects using the prototype property of the function constructor. When adding properties to the constructor it's different to adding properties to an object.

```
FullName.age = 24;
// Result --> undefined
console.log(person.age);

FullName.prototype.age = 24;
// Result --> 24
console.log(person.age);
```

2.6 Static methods

In JavaScript, static methods are bound to a class, not the instance of that class.

Class: Blueprint for creating objects. A class encapsulates data and functions that manipulate data.

Static methods are designed to live only on the constructor in which they were created and static methods cannot be passed down to any children.

Constructor: A special function that creates and initialises an object instance of a class, they get called when an object is created using the new keyword.

```
class Person {
  static changeName(newName) {
    this.newName = newName;
  }
}

constructor({ newName = "Sofia" = {}) {
  this.newName = newName;
}
```

In ES6, we define static methods using the static keyword. To call a static method from a class constructor, we use the class name followed by the . and the static method.

```
className.staticMethodName();

// or

this.constructor.staticMethodName();
```

```
let newPerson = new Person({ newName = "Susanna" });
newPerson.changeName("Franek");
console.log(newPerson.newName);
```

In the example above, you can see that newName of the newPerson is still “Susanna”, it hasn’t changed its value to “Franek”. This is because newPerson. changeName isn’t a function, the function isn’t passed down to the newPerson instance. This is evidence that static methods cannot be passed down to any children.

2.7 Polyfills and transpilers

JavaScript like many programming languages steadily evolve. To make our modern code work on older engines that don't understand the most recent features yet we have two tools: transpilers and polyfills.

Polyfills

In JavaScript, a polyfill is a piece of code that can add and/or update a feature that the engine may lack.

```
// if no such function
if (!Object.create) {
  // implement this instead
  Object.create = function(myObj) {
    function newFunction() { }
    newFunction.prototype = myObj;
    return new NewFunction();
  }
}
```

Some popular polyfill libraries are- core.js and polyfill.io.

Transpilers

In JavaScript, a transpiler is a special piece of software that can translate source code to another source code. It can read and understand modern code and rewrite the code into older syntax that will also work on outdated engines. One of the most popular transpilers is Babel.

Project build systems like Webpack allow us to run a transpiler automatically on every code change so it's easy to integrate transpilers into our development process.

2.8 Understanding JavaScript Summary

In this eBook, we have covered the following topics which I think help with our understanding of JavaScript, applying the fundamentals and using them in real practice.

- Syntax Parser
- Creation Phase
- Creation and Hoisting
- Strict mode
- Function statement vs. function expression
- Functional Programming
- JSON and Object Literals
- Execution Stack
- bind(), call() and apply()
- Closures and Callbacks
- Variable Scope
- The Scope Chain
- By value vs. by reference
- Spread operator
- Inheritance and the Prototype Chain
- Prototype
- Static methods
- Polyfills and Transpilers

Recommend resources

Eloquent JavaScript 3rd Edition (Book) - https://eloquentjavascript.net/

Medium - https://medium.com/

MDN Web Docs - https://developer.mozilla.org/en-US/docs/Web/JavaScript

Thank you for reading my eBook.

Happy coding!