**JavaScript Essentials And Advanced**

* **Essentials**
* **Advanced**

**JavaScript Essentials**

JavaScript is a huge topic, with so many different features, styles, and techniques to learn, and so many APIs and tools built on top of it. This module focuses mostly on the essentials of the core language, plus some key surrounding topics — learning these topics will give you a solid basis to work from.

[**Variables**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.1_variables)

Learning outcomes:

* Understand what variables are and why they are so important in programming generally, not just JavaScript.
* Declaring variables with let and initializing them with values.
* Reassigning variables with new values.
* Creating constants with const.
* The difference between variables and constants, and when you would use each one.
* Understand variable naming best practices. If not explicitly covered, all examples should show good variable naming pratices in action.
* The different types of value that can be stored in variables — strings, numbers, booleans, arrays, and objects.

Resources:

* [Storing the information you need — Variables](https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/Variables)

[**Math**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.2_math)

Learning outcomes:

* Basic number operations in JavaScript — add, subtract, multiply, and divide.
* Understand that numbers are not numbers if they are defined as strings, and how this can cause calculations to go wrong.
* Converting strings to numbers with Number().
* Operator precedence.
* Incrementing and decrementing.
* Assignment operators, e.g. addition assignment and subtraction assignment.
* Comparison operators.
* Basic Math object methods, such as Math.random(), Math.floor(), and Math.ceil().

Resources:

* [Basic math in JavaScript — numbers and operators](https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/Math)
* [Numbers and dates](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Numbers_and_dates)

[**Text**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.3_text)

Learning outcomes:

* Creating string literals.
* Understand the need for matching quotes.
* String concatenation.
* Escaping characters in strings.
* Template literals:
  + Using variables in template literals.
  + Multiline template literals.
* String manipulation using common properties and methods such as:
  + length.
  + toString().
  + includes().
  + indexOf().
  + slice().
  + toLowerCase() and toUpperCase().
  + replace().

Resources:

* [Handling text — strings in JavaScript](https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/Strings)
* [Useful string methods](https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/Useful_string_methods)

[**Arrays**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.4_arrays)

Learning outcomes:

* Understand what an array is — a structure that holds a list of variables.
* The syntax of arrays — [a, b, c] and the accessor syntax, myArray[x].
* Modifying array values with myArray[x] = y.
* Array manipulation using common properties and methods, such as:
  + length.
  + indexOf().
  + push() and pop().
  + shift() and unshift().
  + join() and split().
* Advanced array methods such as forEach(), map() and filter().

Resources:

* [Arrays](https://developer.mozilla.org/docs/Learn/JavaScript/First_steps/Arrays)

[**Conditionals**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.5_conditionals)

Learning outcomes:

* Understand what a conditional is — a code structure for running different code paths depending on a test result.
* if ... else ... else if.
* Using comparison operators to create tests.
* AND, OR, and NOT in tests.
* Switch statements.
* Ternary operators.

Resources:

* [Making decisions in your code — conditionals](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/conditionals)

[**Loops**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.6_loops)

Learning outcomes:

* Understand the purpose of loops — a code structure that allows you to do something very similar many times without repeating the same code for each iteration.
* Basic for loops.
* Looping through collections with for ... of.

**Notes**:

* There are many other types of loop in JavaScript that we haven't listed here. It is not necessary (or useful) to understand all of them at this stage. For now, students need to understand the purpose of loops, and the most common types.
* break and continue.

Resources:

* [Looping code](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/Looping_code)

[**Functions**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.7_functions)

Learning outcomes:

* Understand the purpose of functions — to enable the creation of reusable blocks of code that can be called wherever needed.
* Understand that functions are used everywhere in JavaScript and that some are built into the browser and some are user-defined.
* Understand the difference between functions and methods.
* Invoking a function.
* Return values.
* Understand global scope and function/block scope.
* Passing in arguments to function calls.
* Named and anonymous functions.
* Building your own custom functions:
  + Including parameters.
  + Including return values.
* Callback functions — understand that arguments to functions can themselves be functions, and what this pattern is used for.
* Arrow functions.

Resources:

* [Functions — reusable blocks of code](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/Functions)
* [Build your own function](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/Build_your_own_function)
* [Function return values](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/Return_values)
* [Arrow function expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

[**JavaScript object basics**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.8_javascript_object_basics)

Learning outcomes:

* Understand that in JavaScript most things are objects, and you've probably used objects every time you've touched JavaScript.
* Basic syntax:
  + Object literals.
  + Properties and methods.
  + Nesting objects and arrays in objects.
* Using constructors to create a new object.
* Object scope, and this.
* Accessing properties and methods — bracket and dot syntax.
* Object destructuring.

Resources:

* [JavaScript object basics](https://developer.mozilla.org/docs/Learn/JavaScript/Objects/Basics)
* [Object destructuring assignment](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

[**DOM scripting**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.9_dom_scripting)

Learning outcomes:

* Understand what the DOM is — the browser's internal representation of the document's HTML structure as a hierarchy of objects, which can be manipulated using JavaScript.
* Understand the important parts of a web browser and how they are represented in JavaScript — Navigator, Window, and Document.
* Understand how DOM nodes exist relative to each other in the DOM tree — root, parent, child, sibling, and descendant.
* Getting references to DOM nodes, for example with querySelector() and getElementById().
* Creating new nodes, for example with innerHTML() and createElement().
* Adding and removing nodes to the DOM with appendChild() and removeChild().
* Adding attributes with setAttribute().
* Manipulating styles with Element.style.\* and Element.classList.\*.

Resources:

* [Manipulating documents](https://developer.mozilla.org/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents)

[**Events**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.10_events)

Learning outcomes:

* Understand what events are — a signal fired by the browser when something significant happens, which the developer can run some code in response to.
* Event handlers:
  + addEventListener() and removeEventListener()
  + Event handler properties.
  + Inline event handler attributes, and why you shouldn't use them.
* Event objects.
* Preventing default behavior with preventDefault().
* Event delegation.

Resources:

* [Introduction to events](https://developer.mozilla.org/docs/Learn/JavaScript/Building_blocks/Events)

[**Async JavaScript basics**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.11_async_javascript_basics)

Learning outcomes:

* Understand the concept of asynchronous JavaScript — what it is and how it differs from synchronous JavaScript.
* Understand that callbacks and events have historically provided the means to do asynchronous programming in JavaScript.
* Modern asynchronous programming with async functions and await:
  + Basic usage.
  + Understanding async function return values.
  + Error handling with try ... catch.
* Promises:
  + Understand that async/await use promises under the hood; they provide a simpler abstraction.
  + Chaining promises.
  + Catching errors with catch().

Resources:

* [Asynchronous JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous)

[**Network requests with fetch()**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.12_network_requests_with_fetch)

* Understand that [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) is used for asynchronous network requests, which is by far the most common asynchronous JavaScript use case on the web.
* Common types of resources that are fetched from the network:
  + Text content, [JSON](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON), media assets, etc.
  + Data from [RESTful APIs](https://developer.mozilla.org/docs/Glossary/REST). Learn the basic concepts behind REST, including common patterns such as [CRUD](https://developer.mozilla.org/en-US/docs/Glossary/CRUD).
* Understand what single-page apps (SPAs) are, and the issues surrounding them:
  + Accessibility issues behind asynchronous updates, for example, content updates not being announced by screen readers by default.
  + Usability issues behind asynchronous updates, like loss of history and breaking the back button.
* Understand [HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview) basics. You should look at common HTTP methods such as GET, DELETE, POST, and PUT, and how they are handled via fetch().

Resources:

* [Fetching data from the server](https://developer.mozilla.org/docs/Learn/JavaScript/Client-side_web_APIs/Fetching_data)

[**Working with JSON**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.13_working_with_json)

Learning outcomes:

* Understand what JSON is — a very commonly used data format based on JavaScript object syntax.
* Understand that JSON can also contain arrays.
* Retrieve JSON as a JavaScript object using mechanisms available in Web APIs (e.g. Response.json() in the Fetch API).
* Access values inside JSON data using bracket and dot syntax.
* Converting between objects and text using JSON.parse() and JSON.stringify().

Resources:

* [Working with JSON](https://developer.mozilla.org/docs/Learn/JavaScript/Objects/JSON)

[**Libraries and frameworks**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.14_libraries_and_frameworks)

Learning outcomes:

* Understand what third-party code is — functionality written by someone else that you can use in your own project, so you don't have to write everything yourself.
* Why developers use third-party code:
  + Efficiency and productivity: A huge amount of complex functionality is already written for you to use, created in a way that enforces efficient, modular code organization.
  + Compatibility: Reputable framework code is already optimized to work across browsers/devices, for performance, etc. Many frameworks also have systems to output to specific platforms (e.g. Android or iOS) as build targets.
  + Support/ecosystem: Popular frameworks have vibrant communities and help resources to provide support, and rich systems of extensions/plugins to add functionality.
* The difference between libraries and frameworks:
  + A library tends to be a single code component that offers a solution to a specific problem, which you can integrate into your own app (for example, [chart.js](https://www.chartjs.org/) for creating <canvas>-based charts, or [three.js](https://threejs.org/) for simplified 3D GPU-based graphics rendering), whereas a framework tends to be a more expansive architecture made up of multiple components for building complete applications.
  + A library tends to be unopinionated about how you work with it in your codebase, whereas a framework tends to enforce a specific coding style and control flow.
* Why should you use frameworks?
  + They can provide a lot of functionality and save you a lot of time.
  + A lot of companies use popular frameworks such as React or Angular to write their applications, therefore a lot of jobs list frameworks as requirements for applicants to have.
* Why is a framework not always the right choice? A framework:
  + Can easily be overkill for a small project — you might be better off writing a few lines of vanilla JavaScript to solve the problem or using a tailored library.
  + Usually adds a lot of JavaScript to the initial download of your application, leading to an initial performance hit and possible usability issues.
  + Usually comes with its own set of custom syntax and conventions, which can introduce a significant additional learning curve to the project.
  + May be incompatible with an existing codebase because of its architecture choice.
  + Will need to be updated regularly, possibly leading to extra maintenance overhead for your application.
  + May introduce significant accessibility issues for people using assistive technologies because of its architecture (for example, SPA-style client-side routing), which will need to be considered carefully.
* How to choose? A good library or framework must:
  + Solve your problems while offering advantages that significantly outweigh any negatives that it brings to the table.
  + Have good support and a friendly community.
  + Be actively maintained — don't choose a codebase that has not been updated for over a year, or has no users.

Resources:

* [Introduction to client-side frameworks](https://developer.mozilla.org/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction)

[**Debugging JavaScript**](https://developer.mozilla.org/en-US/curriculum/core/javascript-fundamentals/#6.15_debugging_javascript)

Learning outcomes:

* Understand the different types of JavaScript errors, for example, syntax errors and logic errors.
* Learn about the common types of JavaScript error messages and what they mean.
* Using browser developer tools to inspect the JavaScript running on your page and see what errors it is generating.
* Using console.log() and console.error() for simple debugging.
* Error handling:
  + Using conditionals to avoid errors.
  + try ... catch.
  + throw.
* Advanced JavaScript debugging with breakpoints, watchers, etc.

Resources:

**Advanced**

## JavaScript Concepts-

1. JavaScript Closures

A closure is an advanced concept that involves a function and any other data the function can access. So, a Closure is a function that uses variables from the outer lexical scope. The interpreter considers any arguments you pass to functions from the global space. If a function only relies on its internal values and parameters, it's not considered a closure. Remember that functions can access values from other external functions considered closures.

The interpreter stores that data in **Heap Memory**, calling the function and knowing the free variables' values. That also means they require more memory and processing power. Closures are robust and have many advantages. They help with data encapsulation. Plus, they also help with removing redundant and maintaining modular code. Let's see an example:

|  |
| --- |
| function createCaffeineAddict(coffeeType) { const addiction = `addicted to ${coffeeType}`; return function getHigh() { const highMessage = `Feeling wired and ${addiction}!`; console.log(highMessage); }; } const espressoJunkie = createCaffeineAddict('espresso'); const latteLover = createCaffeineAddict('latte'); espressoJunkie(); latteLover(); |

2. JavaScrpit Inheritances

You might also have heard the term prototypal inheritance. The Prototype Chain explains that all objects have a private property called "[[Prototype]]" that allows objects to inherit properties from each other.

In JavaScript, objects also inherit methods from other objects, thanks to this. Some data types like Strings, Numbers, and Arrays inherit valuable methods. When searching for a property or a method, the interpreter will try to find a matching name on the object. If it can't find it, it'll also seek the object's property and even the property of the property. That's how it goes until it reaches the end of the chain. Following our example, we can see the inheritance like this:

|  |
| --- |
| function createCaffeineAddict(coffeeType) { const addiction = `addicted to ${coffeeType}`; return function getHigh() { const highMessage = `Feeling wired and ${addiction}!`; console.log(highMessage); }; }  const espressoJunkie = createCaffeineAddict('espresso'); const latteLover = createCaffeineAddict('latte'); espressoJunkie(); latteLover(); |

3. JavaScript Event Loops

Both the browser and Node.js are constantly running a single-threaded event loop. That means they execute only one line of code at a time. It's easier to picture it if you imagine a circle. The browser and Node.js repeat the process, checking for code execution.

Things get spicy here because sometimes developers deliberately queue tasks. So, the browser executes them on the next event. The event loop checks for pending tasks and runs them in a specific order. Thanks to this mechanism, the browser can execute tasks in a non-blocking way, which is handy since modern websites have many things going on.

4. JavaScript Callback Functions

These functions are great for handling asynchronous operations. The interpreter will give you the results of every function in the order they appear, starting from the top of the file and going downwards. However, if a function takes a long time to complete its task, the next one will execute first. That might be different from what you expected when you wrote the functions. You can quickly solve that by passing the first function as a parameter to the next one. And that's a callback function!

|  |
| --- |
| function weightDisplayer(someWeight) { document.getElementById("display").innerHTML = someWeight + " grams"; }  function calculateTotalWeight(arabicaWeight, robustaWeight, myCallback) { let totalWeight = arabicaWeight + robustaWeight; myCallback(totalWeight); }  // Calculate the total weight of 500 grams of Arabica and 250 grams of Robusta coffee beans calculateTotalWeight(500, 250, weightDisplayer); |

You'll often see callback functions where the first involves a lengthy task, usually fetching data from an API. That's why some people use setTimeout(), but we prefer to keep it as simple as possible. But bear in mind that you'll fall into callback hell if you overuse them or nest too many. You can easily avoid that by using promises to accomplish the same results.

5. JavaScript Async/Await And Promises

Both Async and Await are unique keywords that modify functions in JavaScript. "***async****and****await****make****promises****easier to write.*" Promises are an essential JavaScript aspect to understanding Async and Await. They are objects representing a value that will be available in the future. Hence, they say their value is "pending." Now that we established that, let's get back to Async and Await.

Software developers use async to define an asynchronous function. These are perfect for involving many iterations, such as fetching data from an API or reading a file from a disk. Asynchronous functions will automatically return a promise, but you can pause their execution using the *await* keyword. This way, the function will wait for some other promises to resolve. That can improve readability and error handling. Let's keep with our coffee example!

|  |
| --- |
| async function prepareCoffee() { try { const beans = await fetch('https://example.com/api/beans'); const groundBeans = await grindBeans(beans); const brewedCoffee = await brewCoffee(groundBeans);  return brewedCoffee; } catch (error) { console.error(error); } }  const myCoffee = prepareCoffee(); console.log(myCoffee); |

6. JavaScript Functional Programming

There's no chance you haven't heard of functional programming. This trendy programming paradigm encourages using only pure functions. That also means that you must avoid using mutability and side effects. In fact, apart from pure functions, immutability is essential in functional programming.

Following these practices sounds tiresome. Yet, the benefits far outweigh the trouble. You must also embrace high-order functions. Don't worry if you don't know what they are. We'll cover that in a sec. First, check this example of functional programming in JavaScript.

|  |
| --- |
| const coffee = { type: 'Arabica', roast: 'dark', hasCaffeine: true, aroma: { notes: ['chocolate', 'nutty', 'spicy'] } };  // Pure function that returns a new object with a modified aroma const addAromaNote = (coffeeObj, note) => ({ ...coffeeObj, aroma: { ...coffeeObj.aroma, notes: [...coffeeObj.aroma.notes, note] } });  const newCoffee = addAromaNote(coffee, 'fruity'); console.log(newCoffee.aroma.notes); // ['chocolate', 'nutty', 'spicy', 'fruity'] console.log(coffee.aroma.notes); // ['chocolate', 'nutty', 'spicy'] |

7. JavaScript High-Order Functions

A function that takes one or multiple functions as parameters or returns a function is a high-order function. They're pretty standard in functional programming. Like any other function, you can pass them as values, which favors reusability. That also makes your code more concise and declarative. Let's see some examples:

|  |
| --- |
| function brew(coffeeMaker, coffeeType) { return coffeeMaker(coffeeType); }  function makeAmericano(coffeeAmount) { return `Brewing ${coffeeAmount} ml of Americano...`; }  function makeLatte(coffeeAmount) { return `Steaming ${coffeeAmount} ml of milk for Latte...`; }  const result1 = brew(makeAmericano, 200); // returns "Brewing 200 ml of Americano..." const result2 = brew(makeLatte, 300); // returns "Steaming 300 ml of milk for Latte..." |

JavaScript has a few built-in higher-order functions developers use all the time. They help to perform complex operations, and are essential to interact with frameworks like React, Vue, and Angular. Let's see some of the most popular ones!

A. JavaScript Reduce()

Reduce is a powerful method that takes an array of elements to reduce them by applying a function to each element. It accumulates all the elements and returns a single value.

B. JavaScript Map()

The Map function allows you to modify each element of an array returning a new identical array. You can also accomplish this by using for loops or nesting. Map() provides a more elegant way to do it following the functional programming rules.

C. JavaScript Filter()

This function can filter an array according to a particular condition and returns a new array with the elements that passed the condition. Remember that the original stays as is since it returns a new array.

D. JavaScript Sort()

The sort() function allows you to overwrite an array by sorting its elements. If it's an array of integers, it'll sort it in ascending order by default. On the flip side, if it's an array of strings, it will sort it alphabetically. What if you don't want to sort an array in alphabetical or ascending order, you may ask? You can easily sort arrays in non-alphabetical or descending order by combining **sort()** with **reverse()**. So after sorting the list, you have to do listname.reverse() to reverse its order.

8. JavaScript Generators

You can think of generators as special functions that you can pause and resume. Plus, they provide a new way to interact with iterators and regular functions. Instead of producing all values simultaneously, they create them as a sequence on the fly.

That might not sound very clear, so let's show you how they work with a cool example. In JavaScript, you can create functions using the function\* syntax. Then you can use the keyword yield to stop the function and return a value to the user.

|  |
| --- |
| function\* coffeeGenerator() { const coffeeTypes = ['latte', 'cappuccino', 'espresso', 'americano']; for (let i = 0; i < coffeeTypes.length; i++) { yield coffeeTypes[i]; } }  const coffee = coffeeGenerator(); console.log(coffee.next().value); // latte console.log(coffee.next().value); // cappuccino console.log(coffee.next().value); // espresso console.log(coffee.next().value); // americano |

Generators don't produce those values simultaneously. They're much more memory-efficient than JavaScript arrays. Hence, you might want to use them to iterate over large datasets. Remember that you might use them less often while working as a web developer. However, it's vital to have them in your tool belt.

9. JavaScript Hoisting

This concept will definitely blow your mind if you come from another language. Hoisting allows you to declare variables and functions after their assignment. It has this name because it is as if the interpreter hoists those variables and functions to the top of the scope. This way, it executes the code with no errors. You can only take advantage of this using the function and var keywords. If you use cons or let, the interpreter will not hoist the variables or functions you declare. Let's show you how that works with a quick example:

|  |
| --- |
| var coffeeBlend; console.log(coffeeBlend); coffeeBlend = "Dark Roast";  // Output: undefined brewCoffee(); function brewCoffee() { console.log("Brewing a fresh pot of coffee!"); }  // Output: "Brewing a fresh pot of coffee!" |

10. JavaScript IIFEs

Invoked Function Expressions (IIFEs) are functions you don't store in variables. Plus, they don't receive a name, either. Hence, IIFEs just run after you call them. They can be handy and improve your code's quality. By using closures, you avoid declaring variables on the global scope. That's one of their most popular uses. Let's now see a quick example:

|  |
| --- |
| (function(coffeeType) { console.log("Brewing a fresh pot of " + coffeeType + " coffee!"); })("French roast"); // Output: "Brewing a fresh pot of French roast coffee!" |

11. JavaScript Memoization

Memoization is one of the essential topics for building top-performing web apps. On top of that, you're very likely to deal with questions related to it in [tech coding interviews](https://www.wearecapicua.com/blog/it-interview-questions). When building large web applications, software developers use complex functions. As you can imagine, they can take a while to load. Sometimes, they receive many calls to return the same value several times. That can be highly inefficient.

That's when memoization catches the values based on the arguments. This way, when the function receives another call, it gives the result instantly. That's how they improve performance. Memoization is a fundamental topic of Dynamic Programming, so you'll see it a lot in [React.js](https://www.wearecapicua.com/blog/ultimate-guide-react).

|  |
| --- |
| // Define a function to memoize function multiply(x, y) { return x \* y; }  // Define a memoization cache as an object const cache = {};  // Define a memoized version of the function function memoizedMultiply(x, y) { const cacheKey = x + ":" + y; if (cache[cacheKey] !== undefined) { return cache[cacheKey]; } else { const result = multiply(x, y); cache[cacheKey] = result; return result; } }  // Call the memoized function console.log(memoizedMultiply(2, 3)); // Should print "6" console.log(memoizedMultiply(2, 3)); // Should print "6" again (result is already memoized) |

12. JavaScript Currying

Currying transforms a function that receives many arguments into a sequence of functions. These new functions will only receive one argument. That's what currying can do for you! This powerful technique has that name because of **Haskell Brooks Curry**, a famous mathematician and logician, and the concept of currying comes from **Lambda Calculus**. Let's go back to how you can use currying in JavaScript.

|  |
| --- |
| function brewCoffee(beans) { return function(water) { return beans \* water; } } const coffeeMaker = brewCoffee(10); // returns a function that multiplies its argument by 10 (the number of coffee beans) console.log(coffeeMaker(8)); // prints 80 (the amount of water) console.log(coffeeMaker(12)); // prints 120 (the amount of water) |

Final Thoughts

The topics covered in this blog post will give you the foundation to build robust web apps. Plus, they will improve your understanding of the JavaScript code essence. In this manner, you'll be an industry expert, and it will lead you to learn popular frameworks more easily. Make sure you let each concept solidify before you move on to the next one, as they can be hard to grasp if you are unfamiliar. Grab another cup of coffee, and happy developing!