BEGINNING MODERN JavaScript

A Step-By-Step Gentle Guide to Learn JavaScript for Beginners

NATHAN SEBHASTIAN

TABLE OF CONTENTS

Beginning Modern JavaScript

A Step-By-Step Gentle Guide to Learn JavaScript for Beginners

By Nathan Sebhastian

PREFACE

The goal of this book is to provide a gentle step-by-step instructions that will help you see how to develop web applications using JavaScript.

Instead of covering all the theories and concepts of JavaScript, I'll be teaching you only the most important building blocks of the language. We'll cover things like variables, data types, arrays, functions, and objects before learning how to build and manipulate the web using JavaScript.

After finishing this book, you will have a solid foundation in building a front-end application using JavaScript.

Working Through This Book

This book is broken down into 17 short chapters, where each chapter will focus on a specific aspect of JavaScript. The last few chapters will take all you've learned in the previous ones so that you can build a Wizard Form and a To-do List application using JavaScript, HTML, and CSS.

I encourage you to write the code you see in this book and run them so that you have a sense of what web development with

JavaScript looks like. You learn best when you code along with examples in this book.

A tip to make the most of this book: Take at least a 10-minute break after finishing a chapter, so that you can regain your energy and focus.

Also, don't despair if some concept is hard to understand. Learning anything new is hard for the first time, especially something technical like programming. The most important thing is to keep going.

Requirements

No previous JavaScript programming knowledge is required as I will introduce JavaScript from the basics, but you should be somewhat familiar with HTML and CSS to make the most of this book.

Contact

If you need help, you can contact me at nathan@codewithnathan.com.

You might also want to subscribe to my 7-day free email course called Mindset of the Successful Software Developer at https://sebhastian.gumroad.com/l/mindset

The email course would help you find the right path forward as a software developer.

CHAPTER 1: INTRODUCTION

What is JavaScript?

JavaScript was created around April 1995 by a developer named Brendan Eich. At the time, he was working to develop a browser for a company called Netscape. He was told that he only had 10 days to design and code a working prototype of a programming language that could run on the browser.

He needed to create a language that appealed to non-professional programmers who wanted to hack a website from scratch. The reason he was given only 10 days was that Netscape needed to release its browser, which at the time competed with Microsoft.

In the beginning, JavaScript was not as powerful as it is today, but an active community of JavaScript developers kept adding new features to the language.

With the rise of internet-based companies like Google and Facebook, JavaScript began to grow to accommodate the ambitions of these giant internet companies. The language received contributions from software developers everywhere around the world.

A great innovation happened to make JavaScript even more powerful in 2009. A server-side environment named Node.js was released, allowing JavaScript to run on the server side like PHP, Java, Python, Ruby, and many more. It also enabled developers to create a full-stack web application using only JavaScript.

Today, JavaScript is a language that is used virtually everywhere.

Why Learn JavaScript?

There are 4 good reasons why you need to learn JavaScript:

1. JavaScript is the only language that works in the browser
2. It's fairly easy to learn (but hard to master)
3. It's an essential language for making web applications
4. There are many career opportunities when you know how to work with JavaScript

Learning JavaScript opens tremendous opportunities no matter what role you want to get into in the tech industry. You can be a frontend, backend, mobile, or full-stack developer by mastering JavaScript.

Computer Setup

To start programming with JavaScript, you only need two things:

1. A web browser
2. A code editor

We're going to use the Chrome browser to run our JavaScript code, so if you don't have one, you can download it here:

https://www.google.com/chrome/

The browser is available for all major operating systems. Once the download is complete, install the browser on your computer.

Next, we need to install a code editor. There are several free code editors available on the Internet, such as Sublime Text, Visual Studio Code, and Notepad++.

Out of these editors, my favorite is Visual Studio Code, because it's fast and easy to use.

Installing Visual Studio Code

Visual Studio Code or VSCode for short is a code editor application created for the purpose of writing code. Aside from being free, VSCode is fast and available on all major operating systems.

You can download Visual Studio Code here:

https://code.visualstudio.com/

When you open the link above, there should be a button showing the version compatible with your operating system as shown below:

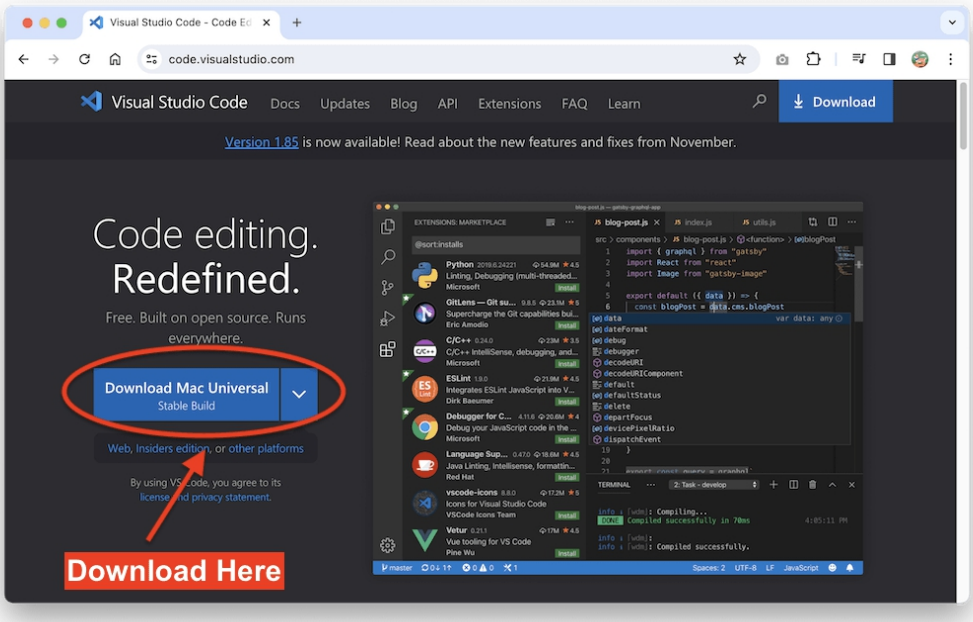


Figure 1. Install VSCode

Click the button to download VSCode, and install it on your computer.

Now that you have a code editor installed, the next step is to create your first JavaScript application.

Your First JavaScript Application

It's time to run your first JavaScript program. First, create a folder on your computer that will be used to store all files related to this book. You can name the folder 'beginning_javascript'.

Next, open the Visual Studio Code, and select *File* > *Open Folder...* from the menu bar. Select the folder you've just created earlier.

VSCode will load the folder and display the content in the Explorer sidebar, it should be empty as we haven't created any files yet.

To create a file, right-click anywhere inside the VSCode window and select *New Text File* or *New File...* from the menu. Once the file is created, press Ctrl+S or Command+S to save the file. Name that file as `index.html`.

The next step is to put some content in the HTML file. You can write the content below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="script.js" defer></script>
</head>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

The HTML code above is just a standard template for a new HTML document, in which we set the document type with `DOCTYPE` and the language with `lang`.

Inside the `<head>` tag, we defined basic information about this document, such as the character encoding with `charset`, the viewport and default zoom level to 1.0, the title of the document, and the `<script>` tag to load a JavaScript file.

Notice that the `<script>` tag above contains the `defer` attribute. This attribute is added to ensure that the script will be

processed after the HTML document has been loaded by the browser.

Inside the body tag, we just add a heading `<h1>` tag with the 'Hello World!' text.

In your VSCode, you should see different parts of your code highlighted with different colors. This is a feature of the text editor called *syntax highlighting*, and it's really useful to help you distinguish different parts of the code.

Save the changes to the file, and the next step is to run this file using a local server.

Install Live Server Extension

VSCode extensions are small programs created to enhance the capability of VSCode. One extension that's very useful is the Live Server extension, which allows developers to run a local server to test their code.

We're going to use the Live Server extension to run our HTML file. To install extensions, click the Extension icons that look like a puzzle of blocks or press Ctrl+Shift+X or Command+Shift+X.

You should see the extension sidebar. At the top of the sidebar, type in 'live server' and select the one that's created by Ritwick Dey. See the picture below:

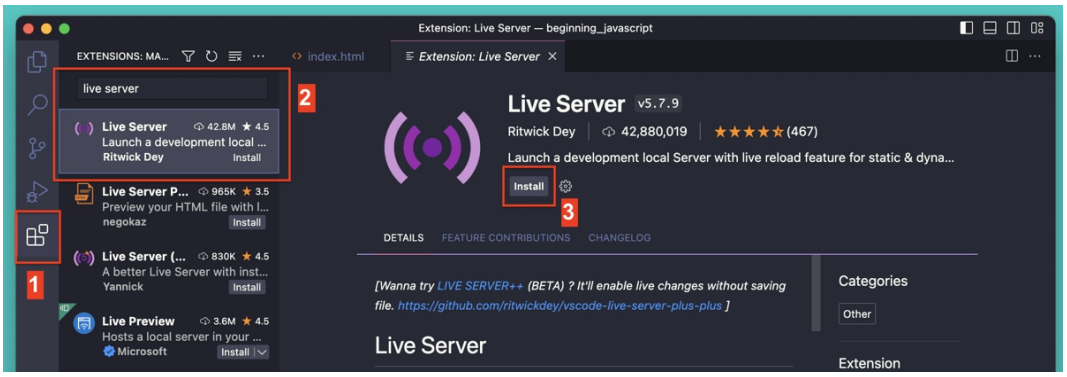


Figure 2. Install Live Server

Explanation of the image above:

1. Click the Extensions icon in the Activity Bar
2. Search for 'live server' and select the one created by Ritwick Dey
3. Install the extension

Once installed, go back to the Explorer tab by clicking on the Files icon at the top of the Activity Bar.

Here, you need to right-click on the `index.html` file and select *Open with Live Server* from the context menu.

The Live Server should start a local server and open your browser automatically as follows:

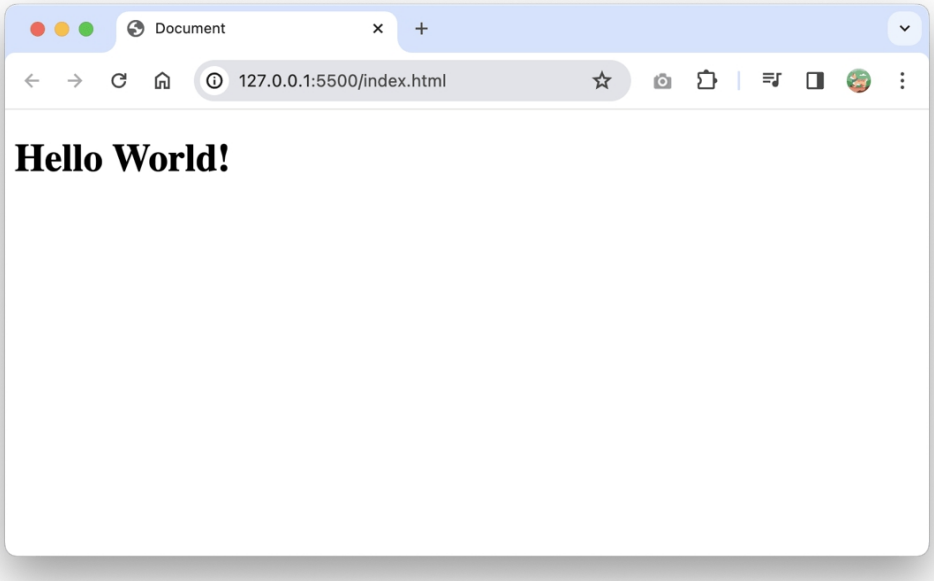


Figure 3. Live Server Running

If you see the image above, congratulations! You now have a local server to help you develop your JavaScript application.

Creating the JavaScript File

Now that we have a local server running, the last thing we need is to create a JavaScript file and print something to the console to know that it's working correctly.

In your VSCode window, right-click on the Explorer sidebar and write a new file named `script.js`.

Inside the file, write the following code:

```
console.log('Learning JavaScript!');
```

Live Server automatically reloads the browser when it detects any change in your files. Back to the browser, you need to open

the browser console by pressing the Command+Option+J or Control+Shift+J

You should see the words printed in the console as follows:

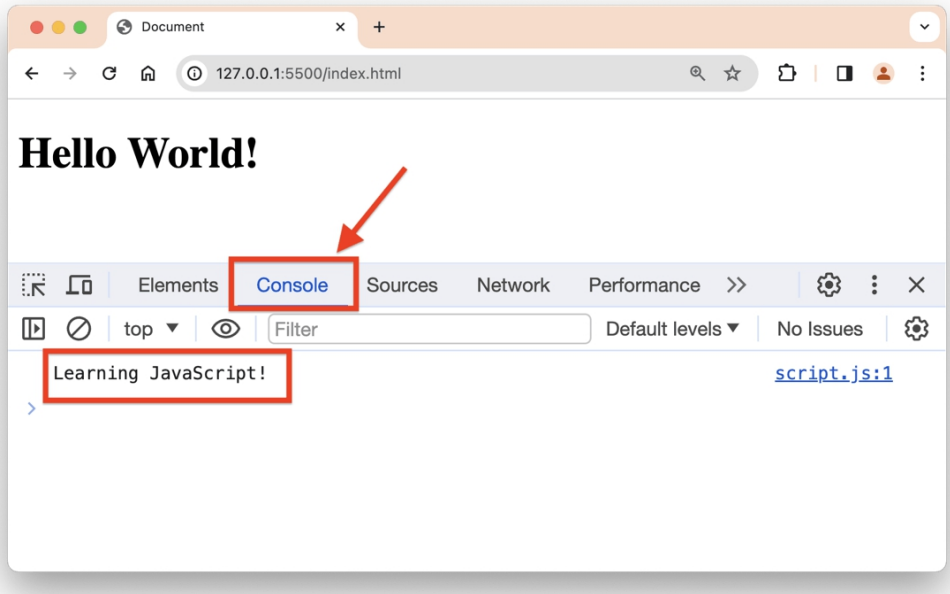


Figure 4. Open Browser Console

With this, you've completed your first JavaScript application. Well done!

The `console.log()` is a keyword used to print output to the console. It consists of the `console` keyword, which is an object that gives access to the browser console.

The `log` keyword is a method used to instruct JavaScript to print a message. By now, you must be wondering what is an object and a method.

Don't worry! We're going to learn more about functions and objects later. For now, just remember that the `console.log()`

keyword is used to print things to the console. This keyword will be used frequently in the coming chapters.

Writing Comments

In programming, comments are text we use to communicate the context of the code written in the file.

To write a comment in JavaScript, you need to add two forward slashes // before the comment as shown below:

```
// This is a comment
// This is also a comment

// Below print two lines of statements
console.log("Hello World!");
console.log("I'm learning JavaScript");
```

Comments are ignored by the language processor, so you can use comments to disable some code without having to delete that code.

The code below shows how to disable the second print statement:

```
console.log("Hello World!");
// console.log("I'm learning JavaScript");
```

Summary

In this chapter, we've explored the origins of JavaScript, installed the required tools to code in JavaScript, and run our first JavaScript application.

We've also learned how to write comments in our JavaScript file. In the next chapter, we're going to learn about variables.

CHAPTER 2: JAVASCRIPT VARIABLES

Before explaining what a variable is, I want you to change the code you've written in the `script.js` file.

Change the code in that file as follows:

```
let message = "Learning JavaScript!"
console.log(message)
```

Go back to the browser console, and you'll see the same output as when you write the 'Learning JavaScript!' message directly inside the `console.log()` function. How can this be?

This is because the message declared in the code above is a *variable*.

In programming, a variable is simply a name that you give to a value so that you can access that value later. You can think of a variable as a label that can be tagged to a certain value, so you can refer to the value using the label.

To declare a variable, you need to type the keyword `let` followed by the variable name.

The first line in the code tells JavaScript to associate the `message` variable with the value 'Learning JavaScript!':

```
let message = "Learning JavaScript!"
```

In the second line, JavaScript is instructed to print the value of `message`, and that's exactly what it does.

You can change the value of your variable by re-assigning another value as follows:

```
let message = "Learning JavaScript!"
print(message)
message = "Nice weather!"
print(message)
```

Run the file and you'll see two lines printed as the output:

```
Learning JavaScript!
Nice weather!
```

Variables are used to reference data so that you can use the same data multiple times in your program.

Next, let's see some rules for naming variables in JavaScript.

Variable Naming

JavaScript has a few naming rules that you need to know to avoid naming errors.

Variable names can only contain alphabet letters, numbers, and underscores (_). This means you can name your variable `message`, `message_1`, `message_2`.

The first character of the variable name must not be a number. `message_1` is okay. `1_message` is not.

You can't use reserved keywords such as `console` because they are used by JavaScript to do certain things. There are many other keywords used by JavaScript that you'll learn in the following sections such as `if`, `for`, and `while`.

Variable names are case-sensitive, which means `Message`, `MESSAGE`, and `message` can be used to create three different variables. But of course, I don't recommend using similar names as it causes confusion.

Sometimes, you need more than one word to declare a variable name. JavaScript has two naming conventions that are used worldwide:

1. `camelCase`
2. `snake_case`

Camel case is a naming convention that uses an uppercase letter for the first character for subsequent words. Here's an example:

```
let myAwesomeVariable
```

The snake case naming convention uses an underscore to separate words. Here's an example:

```
let my_awesome_variable
```

Both are acceptable naming conventions, but you should stick to one of them in your code to avoid confusion.

Constant Variables

There are times when you need to store a value that must not be changed in a variable.

A constant variable is a variable that doesn't change its value as long as the program is running. In JavaScript, a constant variable is declared using the `const` keyword.

The following shows how to declare 2 constants in JavaScript:

```
const FILE_SIZE_LIMIT = 2000
const MAX_SPEED = 300
```

If you try to reassign a constant variable, you'll get an error:

```
const FILE_SIZE_LIMIT = 2000

FILE_SIZE_LIMIT = 3000 // Error!
```

The naming convention for a constant is to use all uppercase letters, although using lowercase letters also works. The all-uppercase standard makes constants stand out more when you read the code.

Summary

In this chapter, we've explored how we can create variables in JavaScript using the `let` and `const` keywords. We've also learned how to access a variable by passing the variable name.

Learning how to use variables to make a program that does what you want it to do is one of the most important skills you can have as a programmer.

But before you learn more about how to make use of variables, let's learn about data types in JavaScript.

CHAPTER 3: BASIC DATA TYPES

Data types are simply definitions for different types of data known to a programming language.

A data type contains specifications about what you can and can't do with that data.

To show you a brain-friendly example, I'm sure you agree that `2 + 2 = 4`?

Well, JavaScript also agrees with that:

```
console.log(2 + 2);

// Output: 4
```

But what if you try to add a number to letters as shown below?

```
console.log(2 + "ABC");
```

Output:

```
2ABC
```

Adding a number to letters will cause JavaScript to concatenate or join the values together. Adding a number to letters is not the

same as adding a number to another number, and JavaScript looks at the data type to know which operation to perform on the data.

In this section, you're going to learn basic data types that JavaScript has:

- Strings
- Numbers
- Booleans
- Null
- Undefined

You will also see how these different types react to operators like the + operator shown above.

First, let's start with strings.

Strings in JavaScript

Strings are simply data defined as a series of characters.

You've seen an example of string data previously when you call the `console.log()` function to print a message:

```
let message = "Hello, Sunshine!";
console.log(message); // Hello, Sunshine!
```

A string needs to be enclosed in quotations. You can use double quotes or single quotes, but they have to match.

You'll get an error when you use different quotation marks like this:

```
// Invalid or unexpected token
let message = "Hello';
```

You can join two or more strings as one with the plus + symbol. Don't forget to add a space before the next string or you'll get a string without spaces!

```
let message = "Hello " + "and " + "Goodbye!";
console.log(message);

// Output: Hello and Goodbye!
```

When printing a variable's value, you can also add strings in the `console.log()` function directly as follows:

```
let message = "Hello, Dave!";

console.log("The message is: " + message);

// Output: The message is: Hello, Dave!
```

This is particularly useful when you have multiple strings to console.log in one sentence as follows:

```
let name = "John";
let topic = "JavaScript";

console.log(name + " is learning " + topic + " today");

// Output: John is learning JavaScript today
```

Or you can also use the template strings format, which allows you to embed a variable directly inside the string as follows:

```
let name = "John";
let topic = "JavaScript";
```

```
console.log(`${name} is learning ${topic} today`);

// Output: John is learning JavaScript today
```

To use the template strings format, you need to use the backtick (`) character to wrap the string instead of quotations.

The variable is embedded in the string using the dollar symbol and curly brackets as in `${variable}`.

This way, JavaScript knows that you're referencing a variable inside the string.

When you have multiple strings to print in a single line, then the template strings format is more convenient because you don't have to break the string with quotations and concatenations.

Next, strings can also represent numbers. You surround the numbers in quotations as follows:

```
let score = "10" + "30";
console.log(score);

// Output: 1030
```

When you join two string numbers with a + symbol, JavaScript will join the two numbers instead of performing arithmetic addition.

This is how strings work in JavaScript. Let's look at numbers next.

Numbers (Integers and Floats) in JavaScript

Number data types represent different kinds of numbers. There are two types of numbers in JavaScript:

- Integers
- Floats

An integer is a whole number without decimals and fractions. Below, you see examples of two integers x and y:

```
let x = 1;
let y = 2;

console.log(x + y);

// Output: 3
```

On the other hand, floats refer to numbers with decimal points like this:

```
let f = 1.2;
let z = 2.35;

console.log(f + z);

// Output: 3.55
```

To create a float, you need to write a number and use `.` to define the decimal values.

With number types, you can perform arithmetic operations such as addition +, subtraction -, division /, and multiplication * just like how you use a calculator.

Booleans in JavaScript

Boolean is a type that represents `true` and `false` values.

You can think of Booleans as a light switch that can only be in one of two positions: on or off.

So it is with Boolean values in programming languages. They are used when JavaScript needs to make a decision: Go left or go right? Right or wrong?

Here's how you create Boolean values in JavaScript:

```
let on = true;
let off = false;
```

This data type is frequently used when you need to make a decision using a control flow. You'll see why Boolean values are very useful when developing an application later.

Undefined

Undefined is a data type in JavaScript used to represent a variable that hasn't been assigned any value yet.

Anytime you declare a variable without assigning any value, the `undefined` value will be assigned to that variable. For example:

```
let first_name;

console.log(first_name); // undefined
```

You can also assign `undefined` to a variable explicitly as follows:

```
let last_name = undefined;
```

But this is usually not recommended, because JavaScript has another value called `null` which is used to mark a variable as empty.

Null

The `null` value is a special data type that represents an empty or unknown value. Here's how you assign a variable as null:

```
let first_name = null;
```

The code above means that the value of `first_name` is empty or unknown.

At this point, you may be thinking what's the difference between `undefined` and `null`? They seem to serve a similar purpose.

And you are correct. Both `undefined` and `null` are values that represent nothing, and other programming languages usually only have one, and that is `null`.

In JavaScript, the `undefined` value is reserved as the default value when a variable is declared, while `null` means you intentionally assign an "empty" value for the variable.

To summarize, JavaScript treats `undefined` as the "default" empty value and `null` as the "intentional" empty value.

Type Conversion

At times, you might want to convert one data type into another so that the program runs as expected.

For example, suppose you need to convert a string into an integer so you can perform an addition between numbers.

If you have one of the numbers as a string, JavaScript joins them together instead of adding:

```
let x = "7";
let y = 5;

console.log(x + y); // 75
```

To add the two numbers properly, you need to convert the `x` variable into an integer.

Changing the data from one type to another is also known as type conversion or type casting. There are 3 functions frequently used to do type conversion:

- `Number()`
- `String()`
- `Boolean()`

As their name implies, these type conversion functions will attempt to convert any value you specified inside the parentheses to the type of the same name.

To convert a string into an integer, you can use the `int()` function:

```
let x = "7";
let y = 5;

// Convert x to integer
x = Number(x);

console.log(x + y); // 12
```

On the other hand, the `String()` function converts a value of another type to a string. If you type `String(true)`, then you'll get 'true' back.

Passing a value of the same type as the function has no effect. It will just return the same value back.

Type Coercion

In JavaScript, type coercion is a process where a value of one type is implicitly converted into another type.

This is automatically done by JavaScript so that your code won't cause an error. But as you'll see in this section, type coercion can actually cause undesired behavior in the program.

Let's consider what happens when you perform an addition between a `number` and a `string` in JavaScript:

```
console.log(1 + "1");
```

As you've seen in the previous section, JavaScript will consider the number as a string and join the two letters as 11 instead of adding them (`1 + 1 = 2`)

But you need to know that other programming languages don't respond the same way.

Programming languages like Ruby or Python will respond by stopping your program and giving an error as feedback. It will respond with something along the lines of "Cannot perform addition between a number and a string".

But JavaScript will see this and say: "I cannot do the operation you requested **as it is**, but I can do it if the number 1 is converted to a `string`, **so I'll do just that**."

And that's exactly what type coercion is. JavaScript notices that it doesn't know how to execute your code, but it doesn't stop the program and responds with an error.

Instead, it will change the data type of one of the values without telling you.

While type coercion doesn't cause any errors, the output is something you actually don't want either.

Type Coercion Rules

Type coercion rules are never stated clearly anywhere, but I did find some rules by trying various silly codes myself.

It seems that JavaScript will first convert data types to `string` when it finds different data types:

```
1 + "1" // "11"
[1 ,2] + "1" // "1,21"
true + "1" // "true1"
```

But the order of the values matters when you have an object. Writing objects first always returns numeric 1:

```
{ a: 1 } + "1" // 1
"1" + { a: 1 } // "1[object Object]"
true + { a: 1 } // "true[object Object]"
{ a: 1 } + 1 // 1
```

JavaScript can calculate between boolean and numeric types, because boolean values `true` and `false` implicitly have the numeric value of 1 and `0`:

```
true + 1 // 1+1 = 1
false + 1 // 0+1 = 1
[1,2] + 1 // "1,21"
```

Type coercion is always performed **implicitly**. When you assign the value as a variable, the variable type will never change outside of the operation:

```
let myNumber = 1;
console.log(myNumber + "1"); // prints 11
console.log(myNumber); // still prints number 1 and not string
```

You can try to find some more on your own, but you hopefully understand what type coercion is and how it works by now.

Why You Should Avoid Type Coercion

JavaScript developers are generally divided into two camps when talking about type coercion:

- Those who think it's a feature
- Those who think it's a bug

If you ask me, I would recommend that you avoid using type coercion in your code all the time.

The reason is that I've never found a problem where type coercion is required for the solution, and when I need to convert one type into another, it's always better to do so explicitly:

```
let price = "50";
let tax = 5;

let totalPrice = Number(price) + Number(tax);

console.log(totalPrice);
```

Using explicit type conversion functions such as `Number()` and `String()` will make your code clear and transparent. You don't need to guess the correct data type required in your program.

Type coercion is one of the unique features in JavaScript that may confuse beginners, so it's good to clear it up early.

Summary

JavaScript provides five basic data types that you can use in creating a useful program. Each data type has specific traits that define what you can and can't do with the data.

In the next chapter, you're going to learn about operators, which allows you to further manipulate any data defined in your code.

CHAPTER 4: JAVASCRIPT OPERATORS

As the name implies, operators are symbols you can use to perform operations on your data.

You've seen some examples of using the plus + operator to join multiple strings and add two numbers together. Of course, JavaScript has more than one operator as you'll discover in this section.

Since you've learned about data types and conversion previously, learning operators should be relatively easy.

Arithmetic Operators

The arithmetic operators are used to perform mathematical operations like additions and subtractions.

These operators are frequently used with number data types. Here's an example:

```
console.log(10 - 3); // 7
console.log(2 + 4); // 6
```

In total, there are 8 arithmetic operators in JavaScript:

```
// 1. Addition (+)
// Returns the sum between the two operands
console.log(5 + 2); // 7

// 2. Subtraction (-)
// Returns the difference between the two operands
console.log(5 - 2); // 3

// 3. Multiplication (*) operator
// Returns the multiplication between the two operands
console.log(5 * 2); // 8

// 4. Exponentiation (**) operator
// Returns the value of the left operand raised to the power of the right operand
console.log(5 ** 2); // 4

// 5. Division operator
// Returns the sum between the two operands
console.log(5 / 2); // 4

// 6. Remainder operator
// Returns the remainder of the left operand after being divided by the right
operand
console.log(5 % 2); // 1
```

These operators are pretty straightforward. As you've seen in the previous section, the + operator can also be used on strings data to merge multiple strings as one:

```
let message = "Hello " + "human!";
console.log(message); // Hello human!
```

When you add a number and a string, JavaScript will perform a type coercion and treat the number value as a string value:

```
let sum = "Hi " + 89;
console.log(sum); // Hi 89
```

Using any other arithmetic operator with strings will cause JavaScript to return a `NaN` value, which stands for Not a Number. The `NaN` is a number data type.

The Assignment Operator

The next operator to learn is the assignment operator and its variants, which is represented by the equals = sign.

In JavaScript, the assignment operator is used to assign data or a value to a variable.

You've seen some examples of using the assignment operator before, so here's a reminder:

```
// Assign the string value 'Hello' to the 'message' variable
let message = "Hello";

// Assign the Boolean value true to the 'on' variable
let on = true;
```

You may've noticed that the equals sign has a slightly different meaning in programming than in math, and you're correct!

The assignment operator isn't used to compare if a number equals another number in programming.

If you want to do that kind of comparison, then you need to use the equal to == operator. We'll learn more about comparison in the next section.

Assignment operators can also be combined with arithmetic operators, so that you can add or subtract values from the left operand.

There are 6 types of assignment operators available in JavaScript. Among these, only the standard assignment and the addition assignment are used frequently. The rest is nice to know.

Standard Assignment Operator

The assignment operator is used to assign one value to another:

```
let x = 2
let y = 1

// Assign y to x
x = y
console.log(x); // 1
```

Addition Assignment Operator

The addition assignment is used to perform addition on the two operands, and assign the result to the left operand:

```
let x = 2
let y = 1

x += y
console.log(x); // 3
```

This is a shorter way to write `x = x + y`.

Subtraction Assignment Operator

The subtraction assignment is used to perform subtraction on the two operands, and assign the result to the left operand:

```
let x = 2
let y = 1

x -= y
console.log(x); // 1
```

This is a shorter way to write `x = x - y`.

Multiplication Assignment Operator

The multiplication assignment is used to perform multiplication on the two operands, then assign the result to the left operand:

```
let x = 2
let y = 2

x *= y
console.log(x); // 4
```

This is a shorter way to write `x = x * y`.

Division Assignment Operator

The division assignment is used to perform division on the two operands, then assign the result to the left operand:

```
let x = 5
let y = 2

x /= y
console.log(x); // 2.5
```

This is a shorter way to write `x = x / y`.

Remainder Assignment Operator

The remainder assignment is used to perform remainder on the two operands, then assign the result to the left operand:

```
let x = 10
let y = 3

x %= y
console.log(x); // 1
```

This is a shorter way to write `x = x % y`.

Next, we're going to take a look at the comparison operators.

The Comparison Operators

Comparison operators are used to compare two values. The operators in this category will return Boolean values: either `true` or `false`.

The following table shows all comparison operators available in JavaScript:

Name	Operation Example	Meaning
Equal	`x == y`	Returns `true` if the operands are equal
Not equal	`x != y`	Returns `true` if the operands are not equal
Strict equal	`x === y`	Returns `true` if the operands are equal and have the same type
Strict not equal	`x !== y`	Returns `true` if the operands are not equal, or have different types
Greater than	`x > y`	Returns `true` if the left operand is greater than the right operand
Greater than	`x > y`	Returns `true` if the left operand is greater than the right operand
Greater than or equal	`x >= y`	Returns `true` if the left operand is greater than or equal to the right operand
Less than	`x < y`	Returns `true` if the left operand is less than the right operand
Less than or equal	`x ⇐ y`	Returns `true` if the left operand is less than or equal to the right operand

Here are some examples of using these operators:

```
console.log(9 == 9); // true

console.log(9 != 20); // true

console.log(2 > 10); // false
```

```
console.log(2 < 10); // true

console.log(5 >= 10); // false

console.log(10 <= 10); // true
```

The comparison operators can also be used to compare strings like this:

```
console.log("ABC" == "ABC"); // true

console.log("ABC" == "abc"); // false

console.log("Z" == "A"); // false
```

String comparisons are case-sensitive, as shown in the example above.

JavaScript also has two versions of each comparison operator: loose and strict.

In strict mode, JavaScript will compare the types without performing a type coercion.

You need to add one more equal = symbol to the comparison as follows

```
console.log("9" == 9); // true
console.log("9" === 9); // false

console.log(true == 1); // true
console.log(true === 1); // false
```

You should use the strict comparison operators unless you have a specific reason not to.

Logical Operators

The logical operators are used to check whether one or more expressions result in either `true` or `false`.

There are three logical operators that JavaScript has:

1. && AND operator - if any expression returns `false`, the result is `false`
2. || OR operator - if any expression returns `true`, the result is `true`
3. ! NOT operator - negates the expression, returning the opposite.

These operators can only return Boolean values. For example, you can determine whether '7 is greater than 2' and '5 is greater than 4':

```
console.log(7 > 2 && 5 > 4); // true
```

Let's have a little exercise. Try to run these statements on your computer. Can you guess the results?

```
console.log(true && false);

console.log(false || false);

console.log(!true);
```

These logical operators will come in handy when you need to assert that a specific requirement is fulfilled in your code.

The typeof Operator

JavaScript allows you to check the data type by using the `typeof` operator. To use the operator, you need to call it before specifying the data:

```
let x = 5;
console.log(typeof x) //  'number'

console.log(typeof "Nathan") // 'string'

console.log(typeof true) // 'boolean'
```

The `typeof` operator returns the type of the data as a string. The 'number' type represents both integer and float types, the 'string' and 'boolean' represent their respective types.

Summary

In this chapter, we've learned about the operators available in JavaScript. Of all these operators, only a select few are used frequently, while the rest will be used only in specific conditions.

At first, it might be difficult to memorize the operator you need. The key to remembering the operators is practice. The more code you write, the more you internalize the operators. Just keep this chapter as a reference when you need to use operators later.

CHAPTER 5: CONTROL FLOWS

Up until now, the JavaScript code you've written is executed line by line from top to bottom. But what if you want to run some lines of code only when a certain condition is met?

A computer program usually needs to take into account many different conditions that can arise during the program's execution.

This is similar to how a human makes decisions in their life. For example, do you have money to cover the vacation to Japan? If yes, go. If not, then save more money!

This is where control flow comes in. Control flow is a feature in a programming language that allows you to selectively run specific code based on the different conditions that may arise.

Using control flows allows you to define multiple paths a program can take based on the conditions present in your program.

There are two types of control flows commonly used in JavaScript: conditionals and loops.

Conditional Flow

The conditional flow statement is used to run a piece of code only under specific conditions.

There are two ways you can create a conditional flow:

1. `if...else` statement
2. `switch...case` statement

The if···else Statement

The `if` statement allows you to create a program that runs only if a specific condition is met.

The syntax for the `if` statement is as follows:

```
if (condition) {
  // code to execute if condition is true
}
```

Let's see an example. Suppose you want to go on a vacation that requires 5000 dollars.

Using the `if` statement, here's how you check if you have enough balance:

```
let balance = 7000;

if (balance > 5000) {
  console.log("You have the money for this trip. Let's go!");
}
```

Run the code above once, and you'll see the string printed on the terminal.

Now change the value of `balance` to `3000` and you'll get no response.

This happens because the code inside the `if` statement is only executed when the condition is `true`.

After the `if` statement, you can write another line of code below it as follows:

```
let balance = 7000;

if (balance > 5000) {
  console.log("You have the money for this trip. Let's go!");
}
console.log("The end!");
```

The second `console.log()` call above will be executed no matter what value you assign to the `balance` variable.

If you want it to execute only when the `if` condition is met, then put the line inside the curly brackets as well:

```
let balance = 7000;

if (balance > 5000) {
  console.log("You have the money for this trip. Let's go!");
  console.log("The end!");
}
```

Next, suppose you need to run some code only when the `if` statement condition is not fulfilled.

This is where the `else` statement comes in. See the following example:

```
let balance = 7000;

if (balance > 5000) {
  console.log("You have the money for this trip. Let's go!");
} else {
  console.log("Sorry, not enough money. Save more!");
}
console.log("The end!");
```

Now change the value of `balance` to be less than `5000`, and you'll trigger the `else` block in the example.

JavaScript also has the `else if` statement which allows you to write another condition to check should the previous `if` statement condition isn't met.

Consider the example below:

```
let balance = 7000;

if (balance > 5000) {
  console.log("You have the money for this trip. Let's go!");
} else if (balance > 3000) {
  console.log("You only have enough money for a staycation");
} else {
  console.log("Sorry, not enough money. Save more!");
}
console.log("The end!");
```

When the `balance` amount is less than `5000`, the `else if` statement will check if the `balance` is more than `3000`. If it does, then the program will proceed to recommend you do a staycation.

You can write as many `else if` statements as you need, and each one will be executed only if the previous statement isn't met.

Together, the `if..else..else if` statements allow you to execute different blocks of code depending on the condition the program faces.

The switch···case Statement

The `switch` statement is a part of core JavaScript syntax that allows you to control the execution flow of your code.

It's often thought of as an alternative to the `if..else` statement that gives you more readable code, especially when you have many different conditions to assess.

Here's an example of a working `switch` statement. I will explain the code below:

```
let age = 15;
switch (age) {
  case 10:
    console.log("Age is 10");
    break;
  case 20:
    console.log("Age is 20");
    break;
  default:
    console.log("Age is neither 10 or 20");
}
```

First, you need to pass an expression to be evaluated by the `switch` statement into the parentheses. In the example, the `age` variable is passed as an argument for evaluation.

Then, you need to write the `case` values that the `switch` statement will try to match with your expression. The `case` value is immediately followed by a colon (`:`) to mark the start of the case block:

```
case "apple":
```

Keep in mind the data type of the `case` value that you want to match with the expression. If you want to match a `string`, then you need to put a `string`. `switch` statements **won't perform type coercion** when you have a `number` as the argument but put a `string` for the case:

```
switch (1) {
  case "1":
    console.log("Hello World!");
    break;
}
```

The number expression doesn't match the string case value, so JavaScript won't log anything to the console.

The same also happens for boolean values. The number 1 won't be coerced as `true` and the number `0` won't be coerced as `false`:

```
switch (0) {
  case true:
    console.log("Hello True!");
    break;
  case false:
    console.log("Bonjour False!");
    break;
  default:
    console.log("No matching case");
}
```

The switch Statement Body

The `switch` statement body is composed of three keywords:

- `case` keyword for starting a case block
- `break` keyword for stopping the `switch` statement from running the next `case`
- `default` keyword for running a piece of code when no matching `case` is found.

When your expression finds a matching `case`, JavaScript will execute the code following the `case` statement until it finds the `break` keyword. If you omit the `break` keyword, then the code execution will continue to the next block.

Take a look at the following example:

```
switch (0) {
  case 1:
    console.log("Value is one");
  case 0:
    console.log("Value is zero");
  default:
    console.log("No matching case");
}
```

When you execute the code above, JavaScript will print the following log:

```
> "Value is zero"
> "No matching case"
```

This is because without the `break` keyword, `switch` will continue to evaluate the expression against the remaining cases even

when a matching case is already found.

Your switch evaluation may match more than one case, so the `break` keyword is commonly used to exit the process once a match is found.

Finally, you can also put expressions as `case` values:

```
switch (20) {
  case 10 + 10:
    console.log("value is twenty");
    break;
}
```

But you need to keep in mind that the value for a `case` block **must exactly match** the `switch` argument.

One of the most common mistakes when using the `switch` statement is that people think `case` value gets evaluated as `true` or `false`.

The following `case` blocks won't produce the righ results:

```
let age = 5;

switch (age) {
  case age < 10:
    console.log("Value is less than ten");
    break;
  case age < 20:
    console.log("Value is less than twenty");
    break;
  default:
    console.log("Value is twenty or more");
}
```

You need to remember the differences between the `if` and `case` evaluations:

- `if` block will be executed when the test condition **evaluates to true**
- `case` block will be executed when the test condition **exactly matches** the given `switch` argument

When to Use Switch?

The rule of thumb when you consider between `if` and `switch` is this:

> *“You only use* `switch` *when the code is cumbersome to write using* `if`

For example, let’s say you want to write a weekday name based on the weekday number

Here’s how you can write it:

```
let weekdayNumber = 1;

if (weekdayNumber === 0) {
  console.log("Sunday");
} else if (weekdayNumber === 1) {
  console.log("Monday");
} else if (weekdayNumber === 2) {
  console.log("Tuesday");
} else if (weekdayNumber === 3) {
  console.log("Wednesday");
} else if (weekdayNumber === 4) {
  console.log("Thursday");
} else if (weekdayNumber === 5) {
  console.log("Friday");
} else if (weekdayNumber === 6) {
  console.log("Saturday");
} else {
```

```
  console.log("The weekday number is invalid");
}
```

I don't know about you, but the code above sure looks cumbersome to me! Although there's nothing wrong with the code above, you can make it prettier with `switch`:

```
let weekdayNumber = 1;

switch (weekdayNumber) {
  case 0:
    console.log("Sunday");
    break;
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
  default:
    console.log("The weekday number is invalid");
}
```

Loop Flow

As you program an application in JavaScript, you'll often need to write a piece of code that needs to be executed repeatedly.

Let's say you want to write a program that prints the numbers 1 to 10 in the console. You can do it by calling `console.log` 10

times like this:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);

// and so on..
```

This works, but there's a better way to write this kind of repetitive task.

A loop statement is another type of control flow statement used to execute a block of code multiple times until a certain condition is met.

There are two loop statements used in JavaScript:

- The `for` statement
- The `while` statement

Let's learn how to use these statements in practice.

The for Statement

Instead of repeating yourself 10 times to print the numbers 1 to 10, you can use the `for` statement and write just a single line of code as follows:

```
for (let x = 0; x < 10; x++) {
  console.log(x);
}
```

There you go! The `for` statement is followed by parentheses (()) which contain 3 expressions:

- The `initialization` expression, where you declare a variable to be used as the source of the loop condition. Represented as `x = 1` in the example.
- The `condition` expression, where the variable in initialization will be evaluated for a specific condition. Represented as `x < 11` in the example.
- The `arithmetic` expression, where the variable value is either incremented (++) or decremented (--) by the end of each loop.

These expressions are separated by a semicolon (;)

After the expressions, the curly brackets ({}) will be used to create a code block that will be executed by JavaScript as long as the *còndition* èxpression returns `true`.

You can identify which expression is which by paying attention to the semicolon (;) which ends the statement.

```
for ( [initialization]; [condition]; [arithmetic expression]) {
  // As long as condition returns true,
  // This block will be executed repeatedly
}
```

The arithmetic expression can be an increment (++) or a decrement (--) expression. It is run once each time the execution of the code inside the curly brackets end:

```
for (let x = 10; x >= 1; x--) {
  console.log(x);
```

```
}
```

Or you can also use shorthand arithmetic operators like += and -= as shown below:

```
// for statement with shorthand arithmetic expression
for (let x = 1; x < 20; x += 3) {
  console.log(x);
}
```

Here, the x will be incremented by 3 each time the loop is executed.

Once the loop is over, JavaScript will continue to execute any code you write below the `for` body:

```
for (let x = 1; x < 2; x++) {
  console.log(x);
}
console.log("The for loop has ended");
console.log("Continue code execution");
```

When to Use a for Loop

The for loop is useful **when you know how many times** you need to execute a repetitive task.

For example, let's say you're writing a program to flip a coin. You need to find how many times the coin lands on heads when tossed 10 times. You can do it by using the `Math.random` method:

- When the number is lower than `0.5` you need to increment the `tails` counter
- When the number is `0.5` and up you must increment the `heads` counter

```
let heads = 0;
let tails = 0;
for (x = 1; x <= 10; x++) {
  if (Math.random() < 0.5) {
    tails++;
  } else {
    heads++;
  }
}

console.log("Tossed the coin ten times");
console.log(`Number of heads: ${heads}`);
console.log(`Number of tails: ${tails}`);
```

The example above shows where the `for` loop offers the most effective approach.

Now let's see an alternative exercise about coin flips where the `for` loop is not effective:

Find out how many times you need to flip a coin until it lands on heads.

This time, you don't know **how many times** you need to flip the coin. This is where you need to use the `while` loop statement, which you're going to learn next.

The while Statement

The `while` statement or `while` loop is used to run a block of code as long as the condition evaluates to `true`.

You can define the condition and the statement for the loop as follows:

```
while (condition) {
  statement;
```

```
}
```

Just like the `for` loop, the `while` loop is used to execute a piece of code over and over again until it reaches the desired condition.

See the example below:

```
let i = 0;

while (i < 6) {
  console.log(`The value of i = ${i}`);
  i++;
}
```

Here, the `while` loop will repeatedly print the value of `i` as long as `i` is less than `6`. In each iteration, the value of `i` is incremented by 1 until it reaches 6 and the loop terminates.

Keep in mind that you need to include a piece of code that eventually turns the evaluating condition to `false` or the `while` loop will be executed forever. The example below will cause an infinite loop:

```
let i = 0;

while (i < 6) {
  console.log(`The value of i = ${i}`);
}
```

Because the value of `i` never changes, the while loop will go on forever!

When to Use a While Loop

Seeing that both `while` and `for` can be used for executing a piece of code repeatedly, when should you use a `while` loop instead of

`for`?

An easy way to know when you should use `while` is when **you don't know how many times** you need to execute the code.

Back to the coin toss example, there's one case that's perfect for a `while` loop:

Find out how many times you need to flip a coin until it lands on heads.

You also need to **show how many times** you flip the coin until it lands on heads:

```
let flips = 0;
let isHeads = false;

while (!isHeads) {
  flips++;
  isHeads = Math.random() < 0.5;
}

console.log(`It took ${flips} flips to land on heads.`);
```

Here, the condition `isHead = Math.random() < 0.5` simulates the flipping of a fair coin. When the result is `true`, it means the coin landed on heads and the loop will exit.

Because you can't know how many times you need to loop until you get the number 8, you need to use a `while` loop instead of a `for` loop.

Summary

In this chapter, you've learned about the two types of control flow available in JavaScript: the conditional flow and the loop

flow.

Each type has two different ways of controlling the flow: conditional has `if-else` and `switch`, while loop has `for` and `while` statements.

The `if` and `for` statements are usually preferred over the `switch` and `while` statements. You can think of `switch` and `while` as an alternative syntax that you use when `if` and `for` isn't suitable.

CHAPTER 6: FUNCTIONS

A function is simply a section (or a block) of code that's written to perform a specific task.

For example, the type casting function `String()` is used to convert data of another type to a string.

The `log()` method from `console.log()` that we've used in the previous chapters is also a function, but because `log()` is attached to the `console` object, it's called a method.

You'll learn more about objects and methods in the chapter. For now, just know that a function and a method are essentially the same, except that a method is called from an object.

Besides the built-in functions provided by JavaScript, you can also create your own function.

How to Create Your Own Function

Creating a function starts with typing the `function` keyword followed by the function name, a pair of round brackets, and then a pair of curly brackets.

Here's an example:

```
function greet() {
  // function body here
  console.log("Hello!");
}
```

To call a function, you need to specify the function name followed by parentheses:

```
greet(); // Hello!
```

The code inside the function is executed when you call that function.

Function Parameters and Arguments

Parameters are variables used to accept inputs given when the function is called.

You can specify parameters in the function header, inside the parentheses.

The following example shows a function that has one parameter called `name`:

```
function greet(name) {
  // function body
}
```

How you use that `name` parameter inside the function is up to you.

You can use the parameter inside the `print()` function as follows:

```
function greet(name) {
  console.log(`Hello, ${name}!`);
  console.log("Nice weather today, right?");
}
```

Now whenever you need to call the `greet()` function, you need to pass an input to fill in the `name` parameter.

The input you passed to fill a parameter is called an argument, and here's how to do it:

```
greet("Peter");
```

The 'Peter' string inside the parentheses when calling the `greet()` function will be passed as the `name` parameter.

Run the code to receive this output:

```
Hello, Peter!
Nice weather today, right?
```

You can have more than one parameter when defining the function, but you need to split each parameter with a comma as follows:

```
function greet(name, weather) {
  console.log(`Hello, ${name}!`);
  console.log(`It's ${weather} today, right?`);
}

greet("Nathan", "rainy");
```

Output:

```
Hello, Nathan!
It's rainy today, right?
```

When you specify two parameters in the function header, you need to pass two arguments. If you call the function without passing the arguments then the value will be `undefined`.

In the next section, you'll learn how to create parameters with default values, which allows you to call the function without having to pass an argument to it.

But for now, I hope you see the convenience of having parameters. They make your functions more adaptable and reusable by taking different input values to cover a variety of scenarios the function might have.

As shown in the example, the `name` and `weather` parameters allow you to greet many different people in different weathers.

Whether sunny, rainy, or cloudy, just change the `name` and `weather` arguments when you want to greet another person.

Default Parameters

When defining a function, you can set a default value for any parameter in that function.

For example, the `name` parameter in the function below is a default parameter:

```
function greet(name = "Nathan") {
  console.log(`Hello, ${name}!`);
  console.log("Nice weather today, right?");
}
```

Here, the default value 'Nathan' will be used when no value or `undefined` is passed for the `name` parameter.

You can test this by calling the `greet()` function without an argument as follows:

```
greet();
greet("Jack");
```

Output:

```
Hello, Nathan!
Nice weather today, right?

Hello, Jack!
Nice weather today, right?
```

Any function you define can have a mix of default and non-default parameters.

Here's another example of a function that has one default parameter called `name` and one non-default parameter called `weather`:

```
function greet(weather, name = "Nathan") {
  console.log(`Hello, ${name}!`);
  console.log(`It's ${weather} today, right?`);
}

greet("sunny");
```

Output:

```
Hello, Nathan!
It's sunny today, right?
```

Notice that the `weather` parameter was placed in front of the `name` parameter. This is for convenience so that you don't need to specify the default parameter.

If you place the non-default parameter after the default parameter, then you need to pass a value to the `name` parameter to get to the `weather` parameter.

Consider the example below:

```
function greet(name = "Nathan", weather) {
  console.log(`Hello, ${name}!`);
  console.log(`It's ${weather} today, right?`);
}

greet(undefined, "sunny");
```

To pass an argument to the `weather` parameter, we need to pass `undefined` or any value for the `name` parameter first.

This is why it's better to specify non-default parameters in front of default parameters.

Default Parameters and Null

Back in Section 2, recall that we briefly explored the difference between `undefined` as the "default" empty value and `null` as the "intentional" empty value.

When you pass `undefined` to a function that has a default parameter, the default parameter will be used:

```
function greet(name = "John"){
  console.log(name);
}

greet(undefined); // John
```

As you can see, JavaScript prints the default parameter value `John` when you pass `undefined` to the function.

But when you pass `null` to the function, the default parameter will be ignored:

```
function greet(name = "John"){
  console.log(name);
}

greet(null); // null
```

This is one of the common mistakes that beginners make when learning JavaScript. When you use the value `null`, JavaScript will think you want that value to be empty, so it doesn't replace the value with the default parameter.

When you use `undefined`, then JavaScript will replace it with the default parameter. You might encounter this issue as you work with JavaScript code in your career, so just keep this in mind.

The return Statement

A function can also have a `return` statement inside the code block. A `return` statement is used to return a value back to the caller.

For example, the following function returns the sum of two values:

```
function sum(a, b) {
  return a + b;
}

let result = sum(3, 2);
console.log(result); // 5
```

The value returned by a function can be assigned to a variable for further operation. You can add the `return` statement

anywhere inside the function.

When JavaScript reaches the `return` statement, it skips further code written inside the function block and goes back to where you call the function.

The following function has two return statements:

```
function checkAge(age) {
  if (age > 18) {
    return "You may get a car license";
  }
  return "You may not get a car license yet";
}

console.log(checkAge(20));
console.log(checkAge(15));
```

Output:

```
You may get a car license
You may not get a car license yet
```

When we call the `checkAge()` function the first time, the value of `age` argument is greater than 18, so JavaScript executes the `return` statement inside the `if` block.

The second time we called the function, the `if` condition wasn't met, so JavaScript executed the `return` statement under the `if` block instead.

You can also stop a function execution and return to the caller by specifying the `return` statement without any value:

```
function greet() {
  console.log("Hello!");
  return;
  console.log("Good bye!");
```

```
}

greet()
```

Output:

```
Hello!
```

Here, the `return` statement is called between the `console.log()` calls.

JavaScript executes the first `console.log()` call, then skips the rest of the code. The 'Good bye!' string isn't printed.

Variable Scope

Now that you're learning about functions, it's a good time to talk about variable scope.

A variable declared inside a function can only be accessed from that function. This is because that variable has a local scope.

On the other hand, a variable declared outside of any block is known as a global variable because of its global scope.

These two scopes are important because when you try to access a local variable outside of its scope, you'll get an error. For example:

```
function greet() {
  let myString = "Hello World!";
}

greet();
console.log(myString);
```

When you run the code above, JavaScript responds with an error:

```
ReferenceError: myString is not defined
```

This is because the `myString` variable is declared inside the `greet()` function, so you can't access that variable outside of it. It doesn't matter even if you called that function before accessing the variable.

Meanwhile, a global variable can be accessed from anywhere, even inside a function:

```
let myString = "Hello World!";

function greet() {
  console.log(myString);
}

greet(); // Hello World!
```

Here, the `greet()` function is able to access the `myString` variable declared outside of it.

Keep in mind that this applies only to variables declared using `let` and `const`.

Next, you can also define a local variable with the same name as the global variable without overwriting it.

Here's an example:

```
let myString = "Hello World!";

function greet() {
  let myString = "Morning!";
  console.log(myString);
}
```

```
greet();  // Morning!
console.log(myString); // Hello World!
```

When you call the `greet()` function, a local variable called `myString` is assigned the string 'Morning!'.

Outside of the function, the global variable that's also called `myString` still exists, and the value isn't changed.

JavaScript considers the local scope variable to be a different variable. When you declare the same variable inside a function, any code inside the function will always refer to the local variable.

In practice, you rarely need to declare the same variable in different scopes:

1. Any variable declared outside a function shouldn't be used inside a function without passing them as parameters.
2. A variable declared inside a function should never be referred to outside of that function

Keep this in mind when you write JavaScript functions.

The Rest Parameter

The rest parameter is a parameter that can accept any number of data as its arguments. The arguments will be stored as an array.

You can define a rest parameter in the function header by adding triple dots ... before the parameter name.

Here's an example of creating a function that has a variable length argument:

```
function printArguments(...args){
    console.log(args);
}
```

When calling the `printArguments()` function above, you can specify as many arguments as you want:

```
function printArguments(...args){
    console.log(args);
}

printArguments("A", "B", "C");
// [ 'A', 'B', 'C' ]
printArguments(1, 2, 3, 4, 5);
// [ 1, 2, 3, 4, 5 ]
```

Keep in mind that a function can only have one rest parameter, and the rest parameter must be the last parameter in the function.

You can use a rest parameter when your function needs to work with an indefinite number of arguments.

Arrow Function Syntax

The arrow function syntax allows you to write a JavaScript function with a shorter, more concise syntax.

When you need to create a function in JavaScript, the primary method is to use the `function` keyword followed by the function

name as shown below:

```
function greetings(name) {
  console.log(`Hello, ${name}!`);
}

greetings("John"); // Hello, John!
```

The arrow function syntax allows you to create a function expression that produces the same result as the code above.

Here's the `greetings()` function using the arrow syntax:

```
const greetings = (name) => {
  console.log(`Hello, ${name}!`);
};

greetings("John"); // Hello, John!
```

When you create a function using the arrow function syntax, you need to assign the expression to a variable so that the function has a name.

Basically, the arrow function syntax looks as follows:

```
const fun = (param1, param2, ...) => {
  // function body
}
```

In the code above,

- `fun` is the variable that holds the function. You can call the function as `fun()` later in your code.
- `(param1, param2, ...)` are the function parameters. You can define as many parameters as required by the function.

- Then you have the arrow ⇒ to indicate the beginning of the function.

The code above is equal to the following:

```
const fun = function(param1, param2, ...) {
  // function body
}
```

The arrow function syntax doesn't add any new ability to the JavaScript language.

Instead, it offers improvements to the way you write a function in JavaScript.

At first, it may seem weird as you are used to the `function` keyword.

But as you start using the arrow syntax, you will see that it's very convenient and easier to write.

Single and Multi-line Arrow Functions

The arrow function provides you a way to write a single line function where the left side of the arrow ⇒ is returned to the right side.

When you use the `function` keyword, you need to use the curly brackets {} and the `return` keyword as follows:

```
function plusTwo(num) {
  return num + 2;
}
```

Using the arrow function, you can omit both the curly brackets and the `return` keyword, creating a single line function as shown below:

```
const plusTwo = (num) => num + 2;
```

Without the curly brackets, JavaScript will evaluate the expression on the right side of the arrow syntax and return it to the caller.

The arrow function syntax also works for a function that doesn't `return` a value as shown below:

```
const greetings = () => console.log("Hello World!");
```

When using the arrow function syntax, the curly brackets are required only when you have a multiline function body:

```
const greetings = () => {
  console.log("Hello World!");
  console.log("How are you?");
};
```

Arrow function Without Round Brackets

The round brackets `()` are used in JavaScript functions to indicate the parameters that the function can receive.

When you use the `function` keyword, the round brackets are always required:

```
function plusThree(num) {
  return num + 3;
}
```

On the other hand, the arrow function allows you to omit the round brackets when you have **exactly one parameter** for the function:

The following code example is a valid arrow function expression:

```
const plusThree = num => num + 3;
```

As you can see, you can remove the round and curly brackets as well as the `return` keyword.

But you still need the round brackets for two conditions:

- When the function has no parameter
- When the function has more than one parameter

When you have no parameter, then you need round brackets before the arrow as shown below:

```
const greetings = () => console.log("Hello World!");
```

The same applies when you have more than one parameter.

The function below has two parameters: `name` and `age`.

```
const greetings = (name, age) => console.log("Hello World!");
```

The arrow syntax makes the round brackets optional when you have a single parameter function.

Convert a Normal Function to an Arrow Function Easily

You can follow the **three easy steps** below to convert a normal function to an arrow function:

1. Replace the `function` keyword with the variable keyword `let` or `const`
2. Add = symbol after the function name and before the round brackets
3. Add ⇒ symbol after the round brackets

The code below will help you to visualize the steps:

```
function plusTwo(num) {
  return num + 2;
}

// step 1: replace function with let / const
const plusTwo(num) {
  return num + 2;
}

// step 2: add = after the function name
const plusTwo = (num) {
  return num + 2;
}

// step 3: add => after the round brackets
const plusTwo = (num) => {
  return num + 2;
}
```

The three steps above are enough to convert any old JavaScript function syntax to the new arrow function syntax.

When you have a single line function, there's a fourth optional step to remove the curly brackets and the `return` keyword as follows:

```
// from this
const plusTwo = num => {
  return num + 2;
};

// to this
const plusTwo = num => num + 2;
```

When you have exactly one parameter, you can also remove the round brackets:

```
// from this
const plusTwo = (num) => num + 2;

// to this
const plusTwo = num => num + 2;
```

But the last two steps are optional. Only the first three steps are required to convert any JavaScript `function` and use the arrow function syntax.

Summary

In this chapter, you've learned how to declare and execute functions in JavaScript.

You've also learned about the two syntaxes that you can use to declare a function: The `function` keyword and the arrow function syntax.

At first, the arrow function syntax might look weird, and you might prefer using the `function` keyword. It's totally fine, as I

also feel that way in the beginning.

But as I wrote hundreds and thousands of functions, I began to prefer the arrow function syntax because of its simplicity.

If you're a complete beginner in JavaScript, you can stick with the `function` keyword as it's still widely used and won't cause any problems. But as you go along in your development journey, try to gradually transition to using the arrow function syntax in your projects.

CHAPTER 7: OBJECTS IN JAVASCRIPT

An object is a special data type that allows you to store more than one value. Every object stores data in a `key:value` pair format.

Let's see an example illustrating the benefit of using objects. Suppose you want to store information about a book in your program.

When you use regular variables, it would look like this:

```
let bookTitle = "Beginning JavaScript";
let bookAuthor = "Nathan Sebhastian";
```

While it works okay, it certainly isn't the best way to store related values. JavaScript doesn't know that the two variables are related. Only you, as the human, knows there's a relation between the two variables.

This is where an object is useful. You can declare a single book object and store the data in a `key:value` format:

```
let myBook = {
  title: "Beginning JavaScript",
```

```
  author: "Nathan Sebhastian",
};
```

An object is declared using the curly brackets {}, and each item inside the brackets is written in the `key:value` format.

An object item is also known as a property, with the *key* as property name and *value* as property value. You can think of the key/value pair as a variable declared in an object.

When you need to add more than one item to an object, you need to separate each item using a comma. To access a property, you can use a dot (.) notation like this:

```
console.log(myBook.title); // Beginning JavaScript
```

You can assign a string or numbers as the key of an item, and you can assign any type of data as the value, including a number and a function:

```
let myBook = {
  title: "Beginning JavaScript",
  author: "Nathan Sebhastian",
  price: 9.99,
  describe: function () {
    console.log(`Book title: ${this.title}`);
    console.log(`Book author: ${this.author}`);
  },
};

// call the function
myBook.describe();
```

In the code above, the `describe` key or property is a method that prints the `title` and `author` value from the object. You call the method by adding parentheses after the method name.

The `this` keyword refers to the context of the code, which is the `myBook` object in this case.

Usually, an object key is something that gives more context to the value it holds. A key must also be unique, so you can't use the same key twice in the same object.

For example, if you have data about a book, you can use object keys such as `title`, `author`, and `price` to help you understand the context of the value stored in each key.

Why a function is called a method when inside an object?

You might wonder, if a method is just a function in an object, then why the additional term? It seems unnecessary.

It's because the term method instantly implies that there's an object that this method belongs to.

When you call a function such as `String()`, you call the function without specifying the object that has that function:

```
String(7)
```

But when you're calling a method, you need to specify the object from which you call the function.

For example, when calling the `describe()` method from the `myBook` object:

```
myBook.describe();
```

The term method gives more information to you as the person who works with the program, allowing you to differentiate between "functions" and "functions in an object" without having to see the code.

The same also applies for properties. The term is used to differentiate between "variables" and "variables in an object".

How to access object values

To access the value of an object, you can use either the dot notation `.` or the square brackets `[]` notation.

Here's an example of using the dot notation to access the object properties:

```
let myBook = {
  title: "Beginning JavaScript",
  author: "Nathan Sebhastian",
};

console.log(myBook.title);
console.log(myBook.author);
```

And here's how you use the square brackets to access the same properties:

```
let myBook = {
  title: "Beginning JavaScript",
  author: "Nathan Sebhastian",
};

console.log(myBook["title"]);
console.log(myBook["author"]);
```

Keep in mind that you need to wrap the property name in quotes like a string, or JavaScript will think you're passing a

variable inside the square brackets.

How to add a new property to the object

You can assign a new property to the object using either the dot notation or the square brackets like this:

```
let myBook = {
  title: "Beginning JavaScript",
  author: "Nathan Sebhastian",
};

// add release year property
myBook.year = 2023;

// add publisher property
myBook["publisher"] = "CodeWithNathan";

console.log(myBook);
```

The `console.log()` above will print the following output:

```
{
  title: 'Beginning JavaScript',
  author: 'Nathan Sebhastian',
  year: 2023,
  publisher: 'CodeWithNathan'
}
```

You can add as many properties as you need to the same object.

How to modify object properties

To modify an existing property, you need to specify an existing object property using either the dot or square brackets notation followed by the assignment operator as follows:

```
let myBook = {
  title: "Beginning JavaScript",
```

```
  author: "Nathan Sebhastian",
};

// change the author property
myBook.author = "John Doe";

console.log(myBook);
```

Output:

```
{
  title: 'Beginning JavaScript',
  author: 'John Doe'
}
```

As you can see, the `author` property value has been changed.

How to delete object properties

To delete a property from your object, you need to use the `delete` operator as follows:

```
let myBook = {
  title: "Beginning JavaScript",
  author: "Nathan Sebhastian",
};

delete myBook.author;

console.log(myBook);
```

Output:

```
{ title: 'Beginning JavaScript' }
```

When you try to access the deleted property, you will get the `undefined` value.

How to check if a property exists in an object

To check if a certain property exists in your object, you can use the `in` operator like this:

```
propertyName in myObject
```

The `in` operator returns `true` if the `propertyName` exists in your object.

See the example below:

```
let person = {
  firstName: "Nathan",
  lastName: "Sebhastian"
}

// check if firstName exists
console.log('firstName' in person); // true

// check if age exists
console.log('age' in person); // false
```

Summary

In this chapter, you've learned about the object data type, which is a special data type that can hold many properties and methods.

Objects are frequently used in JavaScript to perform specific tasks. You've seen one of these in the form of `console.log()`, where we instruct JavaScript to print something to the console.

You've also learned how to add, access, remove, and edit properties and methods of an object.

CHAPTER 8 - ARRAYS

An array is a special object that can be used to hold more than one value. An array can be a list of strings, numbers, booleans, objects, or a mix of them all.

To create an array, you need to use the square brackets `[]` and separate the items using a comma.

Here's how to create a list of strings:

```
let birds = ['Owl', 'Eagle', 'Parrot', 'Falcon'];
```

You can think of an array as a list of items, each stored in a locker compartment:

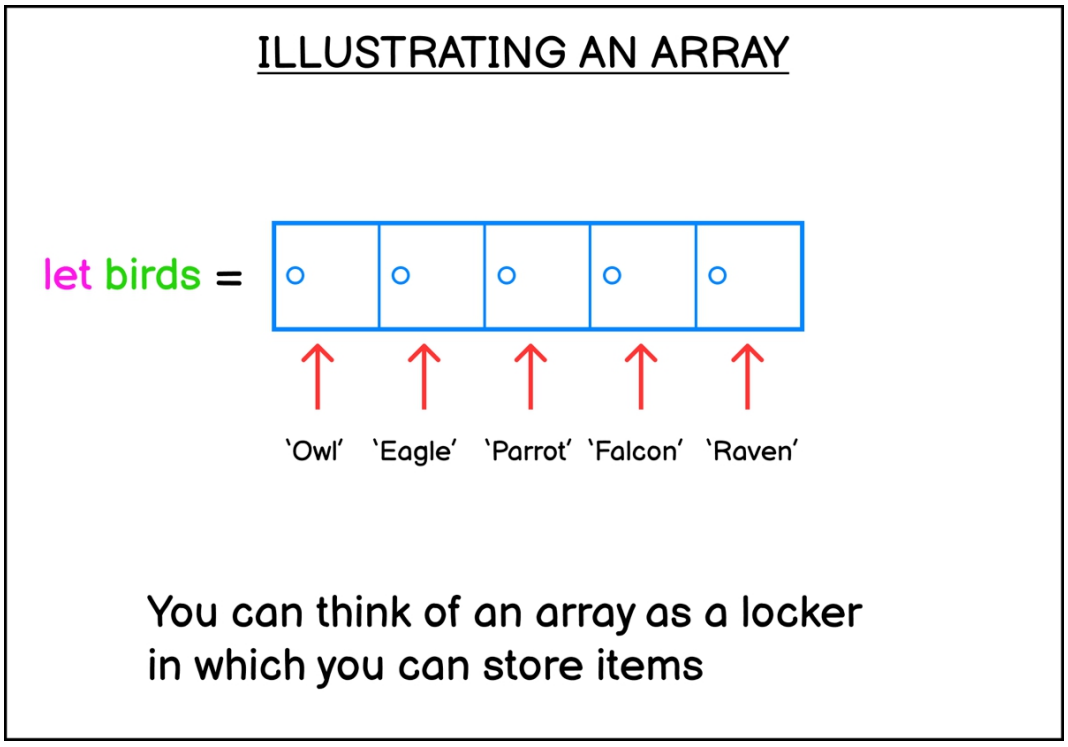


Figure 5. Array Illustration

You can also declare an empty array without any value:

```
let birds = [];
```

An array can also have a mix of values like this:

```
let mixedArray = ['Bird', true, 10, 5.17]
```

The array above contains a string, a boolean, an integer, and a float.

Array Index Position

JavaScript remembers the position of the elements within an array. The position of an element is also called an index number.

Going back to the locker example, you can think of index numbers as the locker numbers. The index number starts from `0` as follows:

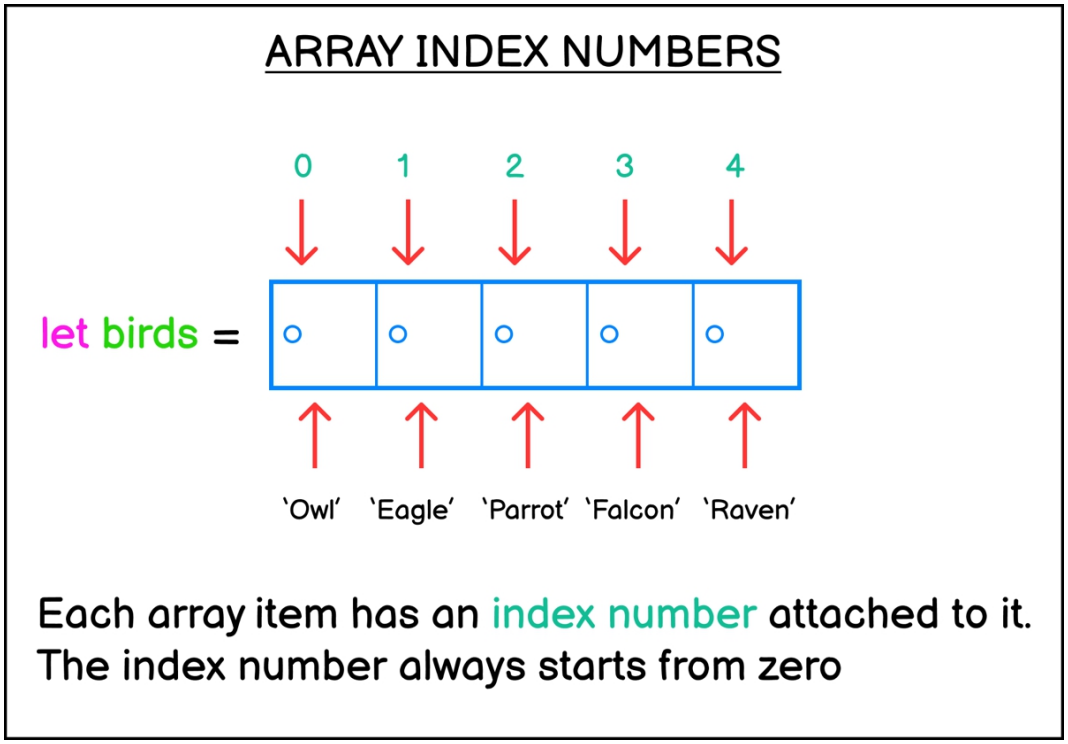


Figure 6. Array Index Illustration

To access or change the value of an array, you need to add the square brackets notation `[x]` next to the array name, where `x` is the index number of that element. Here's an example:

```
// Access the first element in the array
myArray[0];

// Access the second element in the array
myArray[1];
```

Suppose you want to print the string 'Owl' from the `birds` array. Here's how you can do it.

```
let birds = ['Owl', 'Eagle', 'Parrot', 'Falcon'];
```

```
console.log(birds[0]); // Owl
console.log(birds[1]); // Eagle
```

There you go. You need to type the name of the array, followed by the index number wrapped in square brackets.

You can also assign a new value to a specific index using the assignment operator.

Let's replace 'Parrot' with 'Vulture':

```
let birds = ['Owl', 'Eagle', 'Parrot', 'Falcon'];
birds[2] = 'Vulture';

console.log(birds);
// ['Owl', 'Eagle', 'Vulture', 'Falcon']
```

Because the array index starts from zero, the value 'Parrot' is stored at index 2 and not 3.

Array Methods and Properties

Since array is an object, you can call methods and properties that are provided by JavaScript to manipulate the array values.

To get the size of an array, you can access the `length` property:

```
let fishes = ['Salmon', 'Goldfish', 'Tuna'];

console.log(fishes.length); // 3
```

Next, you can use the `push()` method to add an item to the end of the array:

```
let birds = ['Owl', 'Eagle'];

birds.push('Sparrow');
```

```
console.log(birds);
// ['Owl', 'Eagle', 'Sparrow']
```

Another method called `pop()` can be used to remove an item from the end of an array:

```
let birds = ['Owl', 'Eagle', 'Sparrow'];

birds.pop();

console.log(birds);
// ['Owl', 'Eagle']
```

The `unshift()` method can be used to add an item from the front at index 0:

```
let fishes = ['Salmon', 'Goldfish', 'Tuna'];

fishes.unshift('Sardine');

console.log(fishes);
// ['Sardine', 'Salmon', 'Goldfish', 'Tuna']
```

On the other hand, the `shift()` method can be used to remove an item from index 0:

```
let fishes = ['Salmon', 'Goldfish', 'Tuna'];

fishes.shift();

console.log(fishes);
// ['Goldfish', 'Tuna']
```

The `indexOf()` method can be used to find and return the index of an item in the array.

The method will return -1 when the item isn't found inside the array:

```
let fishes = ['Salmon', 'Goldfish', 'Tuna'];

let pos = fishes.indexOf('Tuna');

console.log(pos); // 2
```

To iterate over an array, you can use the `forEach()` method. The method accepts a function that will be executed for each element in the array.

This method will pass the array's element and index position to the function. Here's an example of using the method:

```
let fishes = ['Salmon', 'Goldfish', 'Tuna'];

fishes.forEach(function (element, index) {
  console.log(`${index}: ${element}`);
});
```

The output will be:

```
0: Salmon
1: Goldfish
2: Tuna
```

And that's how you iterate over an array using the `forEach()` method.

Summary

In this chapter, you've learned about the array special data type and how to manipulate the values stored in an array.

By using arrays, you can store data of different types under one variable just like objects, but much simpler as you don't need to add a key for each value.

CHAPTER 9 - THE DOCUMENT OBJECT MODEL (DOM) INTRODUCTION

Before we dive into the lesson, let's recap what we know about the web browser and HTML documents.

When we want to build a web page, we create an HTML document filled with HTML, CSS, and JavaScript.

These three technologies are used together like a team, each playing a different role to build what we see in our web browser.

HTML is a markup language used to define the structure and content of the page. The tags we used like `<head>`, `<body>`, and `<button>` have meanings, and the browser will process these tags, rendering the content of the document to the screen

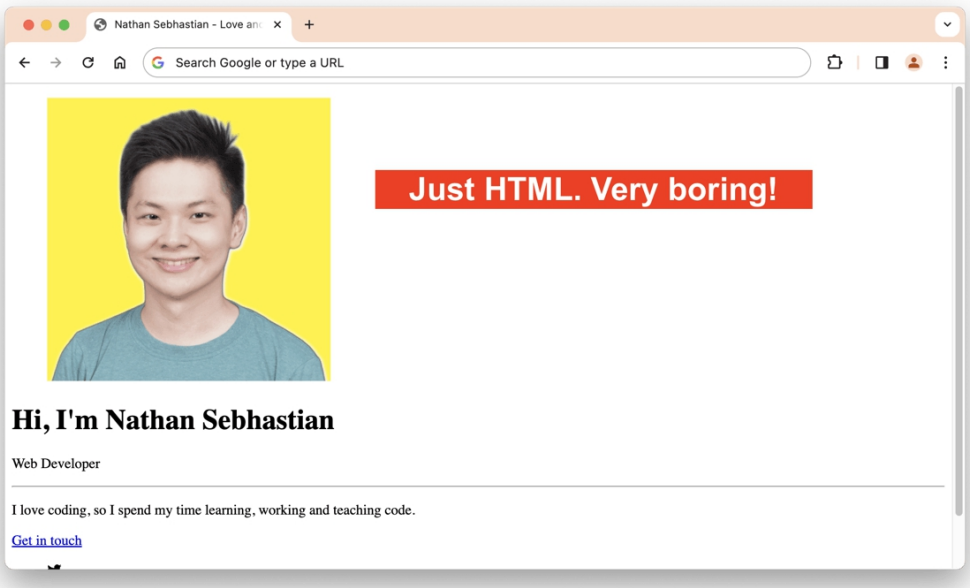


Figure 7. HTML Only Website

But the HTML alone produces a very boring and generic-looking page. CSS was created to enable people to change the way the HTML tags are represented on the browser.

By using CSS, you can make a button displayed in a color of your choice, or a text input box with a rounded border, making them more appealing and modern.

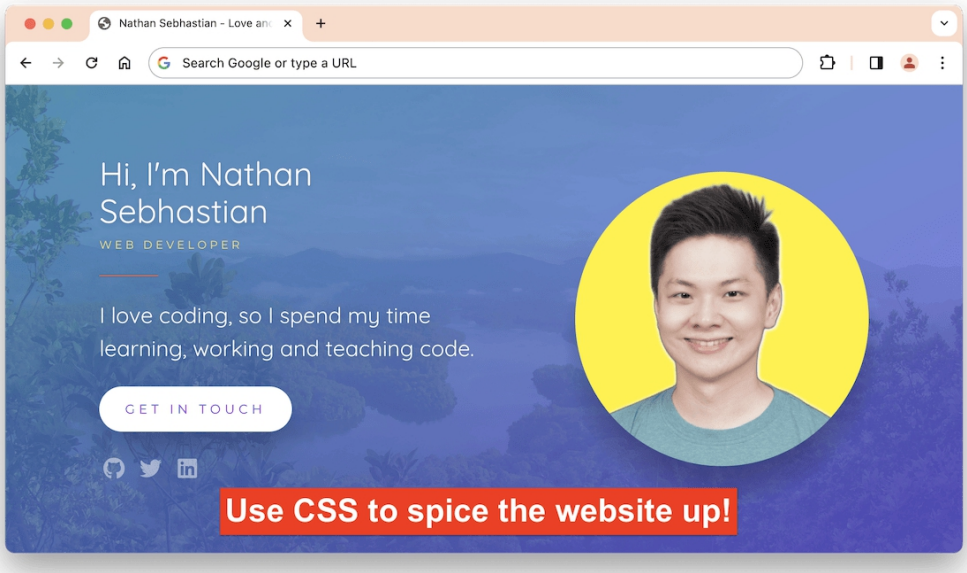


Figure 8. A Website Styled With CSS

HTML and CSS were great partners, but even then, a web page is still just a passive thing. Once the browser process the content and render it, you can't edit the page in any way.

You can't change the background and text color of the page (the dark mode, for example) you can't create cool animations, turn on the microphone or webcam, and retrieve data from another website to display on the page after a click.

JavaScript is the programming language that's supported by the browser, and it fills the gap left open by HTML and CSS.

In short, JavaScript breathes life into static websites, turning them dynamic. It allows a website to interact with the user, respond to their actions, and dynamically update the content without requiring a page reload.

The DOM Explained

So how come JavaScript can do things that HTML and CSS can't do? Well, the answer is that JavaScript has the ability to manipulate the Document Object Model, or the DOM for short.

The DOM is a hierarchical structure composed of objects that make up a web page. Web browsers then expose this DOM so that you can change the page structure, style, and content using JavaScript.

For example, suppose you have the following HTML content in your document:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Hello World!</h1>
  <p>This is a paragraph element</p>
</body>
</html>
```

The DOM graph generated by the HTML document above is as follows. I'm going to explain what the graph means below:

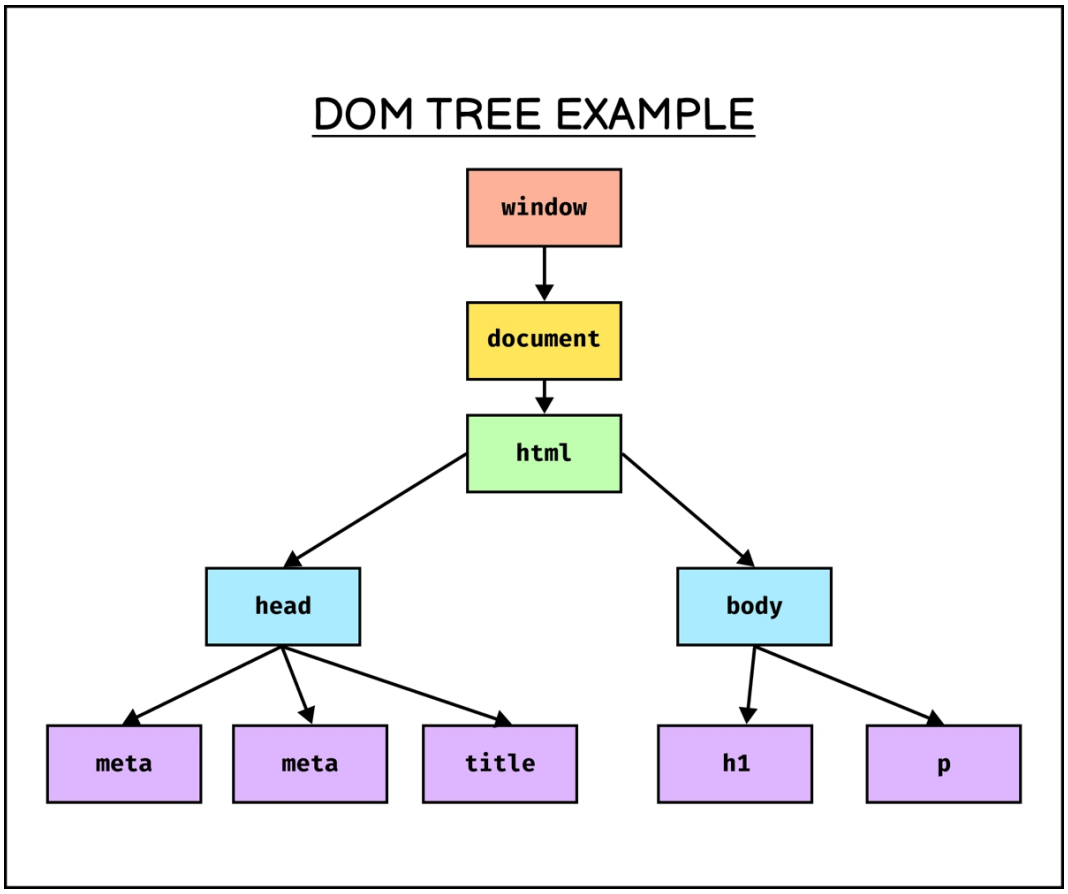


Figure 9. Example DOM Graph

The DOM looks like a tree structure with a set of connected nodes. These nodes are objects that you can access using JavaScript. The `html` node and its children are the content of your document.

For now, let's focus on the `window` and `document` objects.

The Window Object

The `window` object is the root of the DOM tree, and this object is used to instruct the browser to do tasks like:

- Displaying an alert box or a prompt

- Log messages or errors to the console
- Access the browser's local storage
- Access the document object

You can access the window object from the Console by typing `window` and press Enter. The browser would respond by showing you the methods and properties of the object as follows:

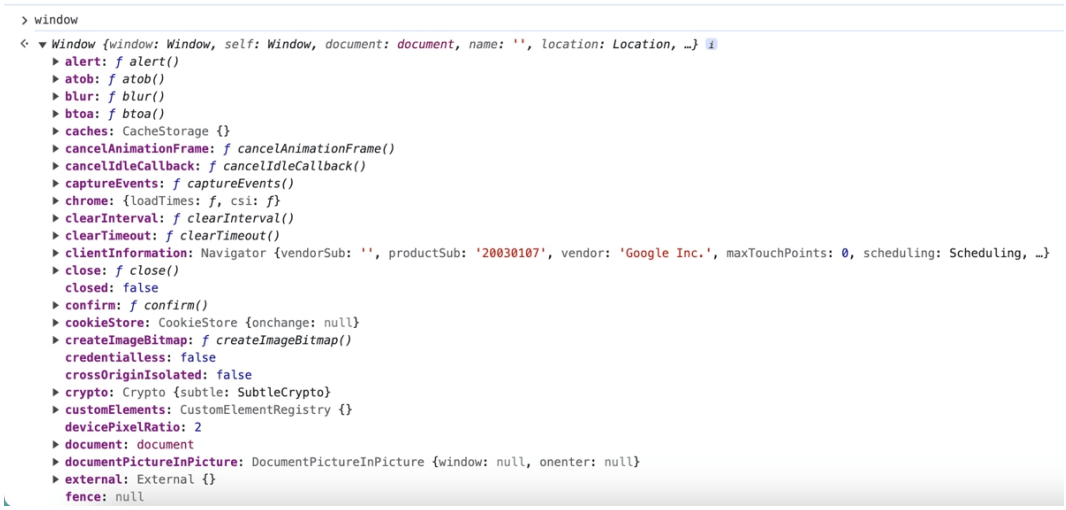


Figure 10. Window Object Properties and Methods

The `window` object provides both child objects and methods that you're going to access to manipulate a web page.

For example, the `window` has the `console` object that you can use to log a message to the console.

Type the code below to the console and hit Enter:

```
window.console.log('Hello World!');
```

You'll see the string 'Hello World!' logged to the console.

The `window` object is a global object, so you can omit it when calling its child object. To access the `console` object, just type console as follows:

```
console.log('Hello World!');
```

The same goes for when you access the `document` object. You can write `document.propertyOrMethod` instead of `window.document.propertyOrMethod`.

I won't go too deep on the `window` object, because as a web developer, you're going to interact more with the `document` object in your daily routine.

The Document Object

The `document` object is the entry point for all HTML elements that you write in your document. This object is also what gives the Document Object Model its name.

It's not called the Window Object Model because we're mostly going to work with the Document object instead of the Window object.

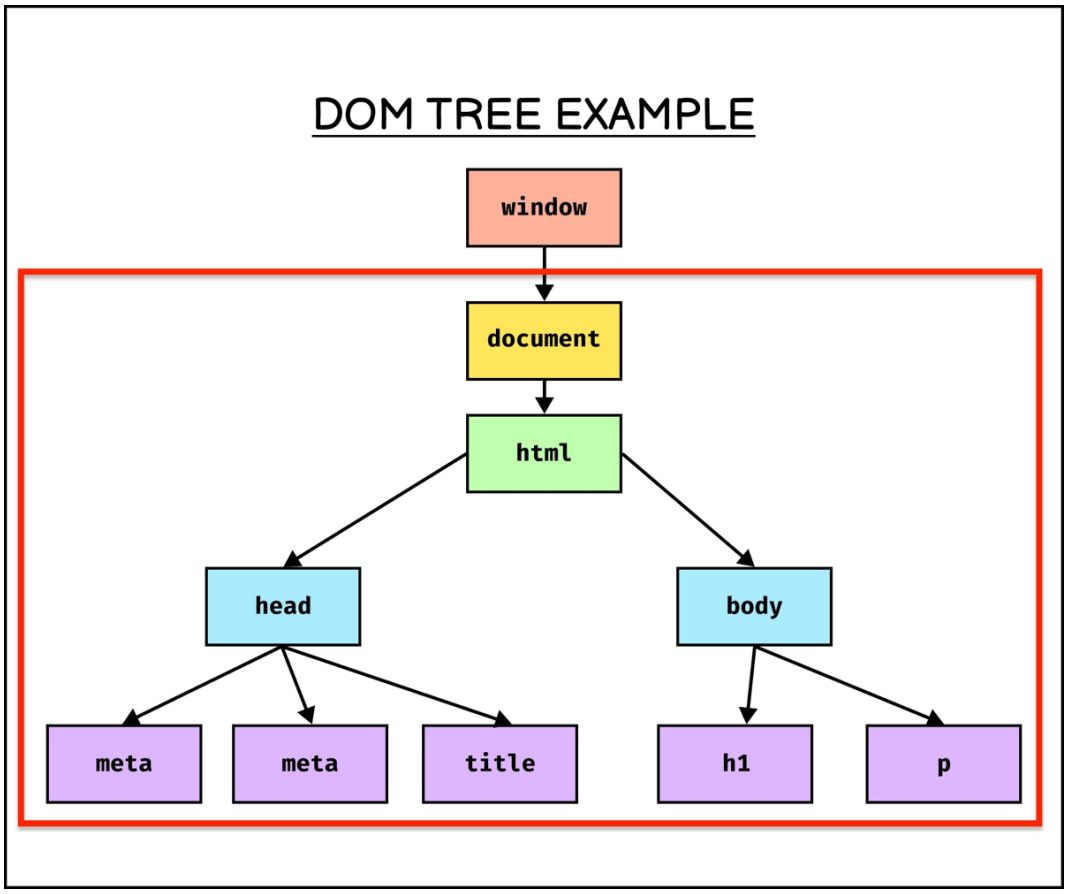


Figure 11. Document Object Model

From the document object you can retrieve any data related to the HTML content using JavaScript. You can get the Document title using the `document.title` property, and the URL using `document.URL`. The referrer is available in `document.referrer`, the domain in `document.domain`.

From the document object you can get the `body` and `head` nodes:

`document.documentElement`: the whole HTML Element node `document.body`: the body Element node `document.head`: the head Element node

You can try to access these from the console as shown below:

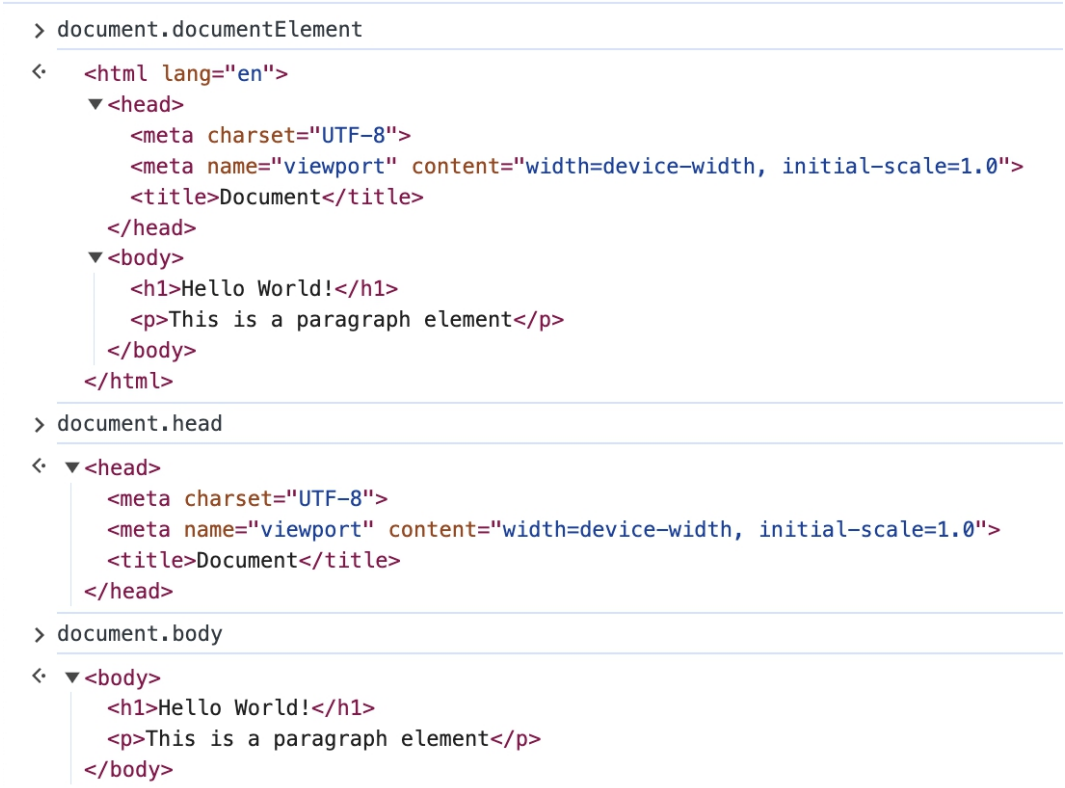


Figure 12. Document Properties

You can see the documentation of all the properties and methods of the document object at https://developer.mozilla.org/en-US/docs/Web/API/Document (note that it might be overwhelming at first)

Of course, you can do much more than just read your Document data. In the next chapter, we're going to start a project to practice manipulating the DOM.

Summary

The DOM is a tree-like structure generated from the HTML file that we created and loaded into the browser.

The browser then exposes this DOM as a set of JavaScript objects that we can access, such as the `window` and `document`

objects.

It’s an important piece of technology that allows you to program the web browser to do specific tasks.

CHAPTER 10: PROJECT 1 - CREATING A WIZARD FORM APPLICATION

We're going to learn how we can use JavaScript to control what's being shown on our web application.

The application we're going to build is a simple Wizard Form. A Wizard Form is a multi-step form designed to ease the filling process for a long and complex form.

By showing only a few inputs at a time, users will feel encouraged to fill in the blank input rather than feeling overwhelmed and potentially abandoning the form.

The Wizard Form has three steps as shown below:

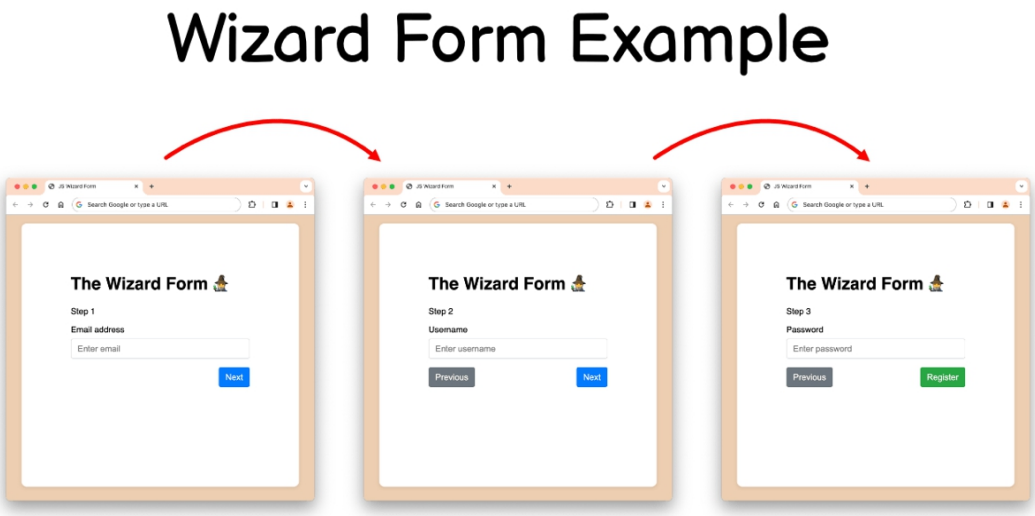


Figure 13. Wizard Form Demo

By clicking the next button, you can jump into the other parts of the form. When you click on the submit button, an alert will be triggered and your inputs will be displayed:

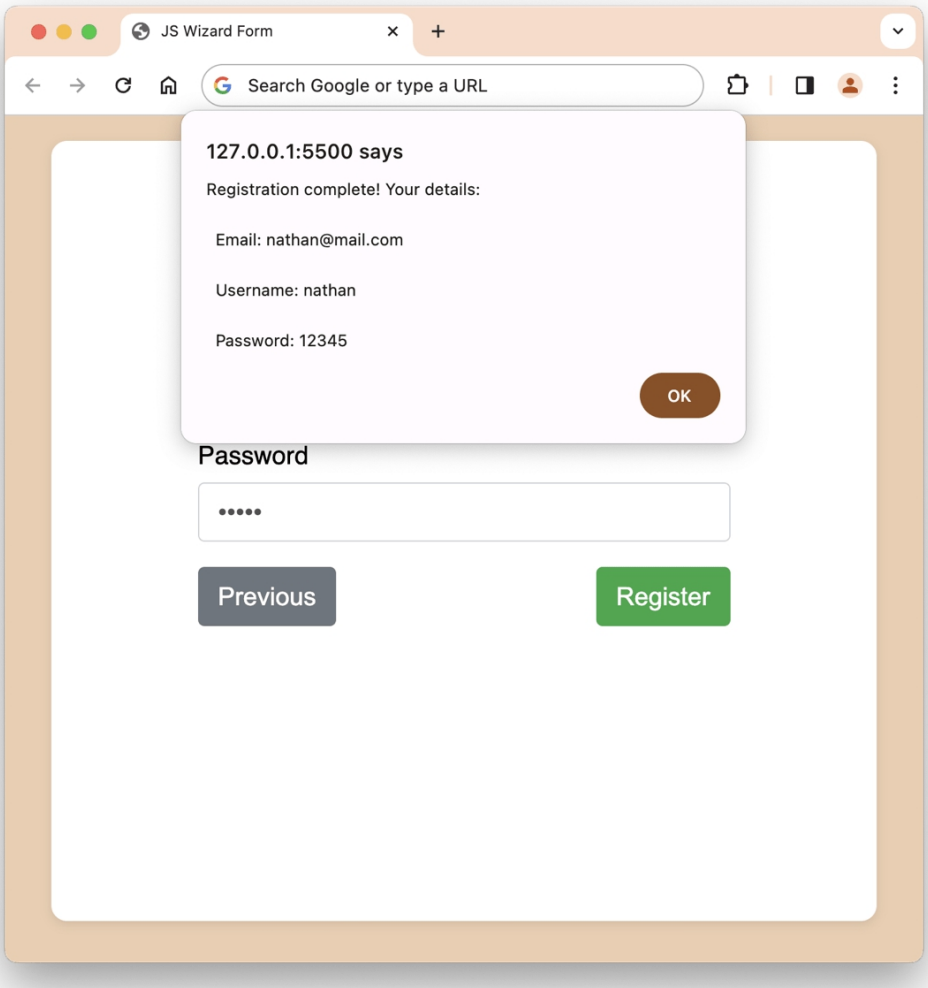


Figure 14. Wizard Form Alert

To keep things simple, the form will only have one input for each step and you don't need to validate the data.

Given the demo above, you basically need to break the form into three `<div>` elements, and then use JavaScript to display them one at a time. Let's start the project in the next section.

Project Setup

To start this project, let's create a new folder to separate the project from the code examples we've written so far.

Let's name the folder `wizard_form` and create three files that we will fill later: `index.html`, `style.css`, and `script.js`.

Putting the HTML and CSS First

When starting a new project, the first step we need to do is to put the HTML and CSS according to the specifications we've been given.

Open the `index.html` file in VSCode and write the basic HTML structure inside:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body></body>
</html>
```

Next, update the `<head>` tag as follows:

1. Edit the title to JS Wizard Form
2. Add the links to the style and script files

The result should look as follows:

```
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>JS Wizard Form</title>
```

```
<link rel="stylesheet" href="style.css" />
<script src="script.js" defer></script>
```

The next step is to write the body of the HTML document. As shown in the demo, there's a container that holds the Wizard Form, so add a container div into the body as follows:

```
<body>
  <div class='container'>
    <h1>The Wizard Form □♂</h1>
    <p>
      Step <span id='currentStep'>1</span>
    </p>
    <form id='wizard-form'></form>
  </div>
</body>
```

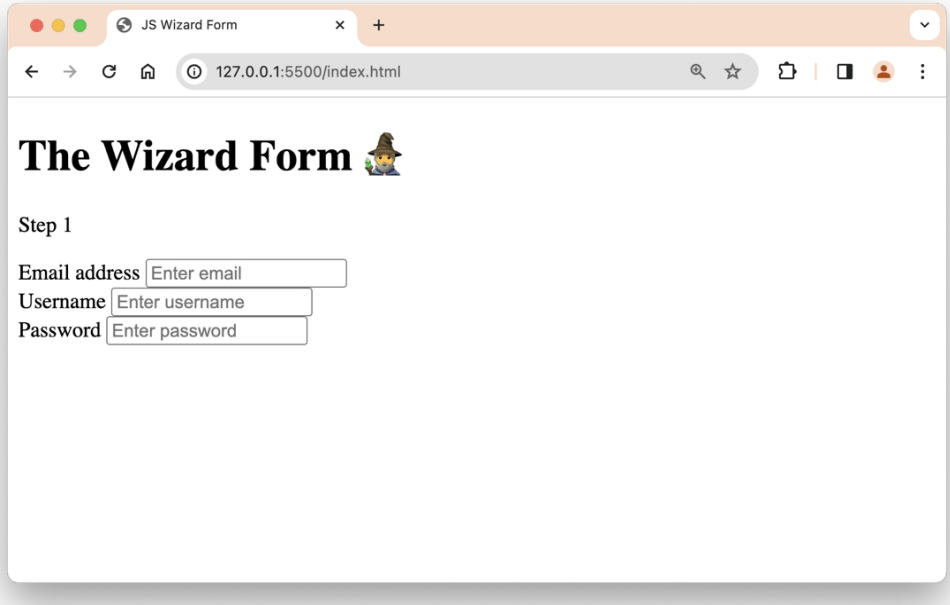
The <form> tag is added as the child of the container div. Now, you need to create the three steps of the Wizard Form.

You can separate the form into three tags:

```
<form id="wizard-form">
  <div id="step1" class="step">
    <div class="form-group">
      <label for="email">Email address</label>
      <input
        class="form-control"
        name="email"
        type="text"
        placeholder="Enter email"
      />
    </div>
  </div>

  <div id="step2" class="step">
    <div class="form-group">
      <label for="username">Username</label>
      <input
        class="form-control"
        name="username"
        type="text"
```

```
        placeholder="Enter username"
      />
    </div>
  </div>

  <div id="step3" class="step">
    <div class="form-group">
      <label for="password">Password</label>
      <input
        class="form-control"
        name="password"
        type="password"
        placeholder="Enter password"
      />
    </div>
  </div>
</form>
```

Now if you open the HTML file using Live Server, you can see the skeleton HTML structure finished:



It's time to add some CSS to beautify this application. Open the `style.css` we created earlier and put the following styles inside.

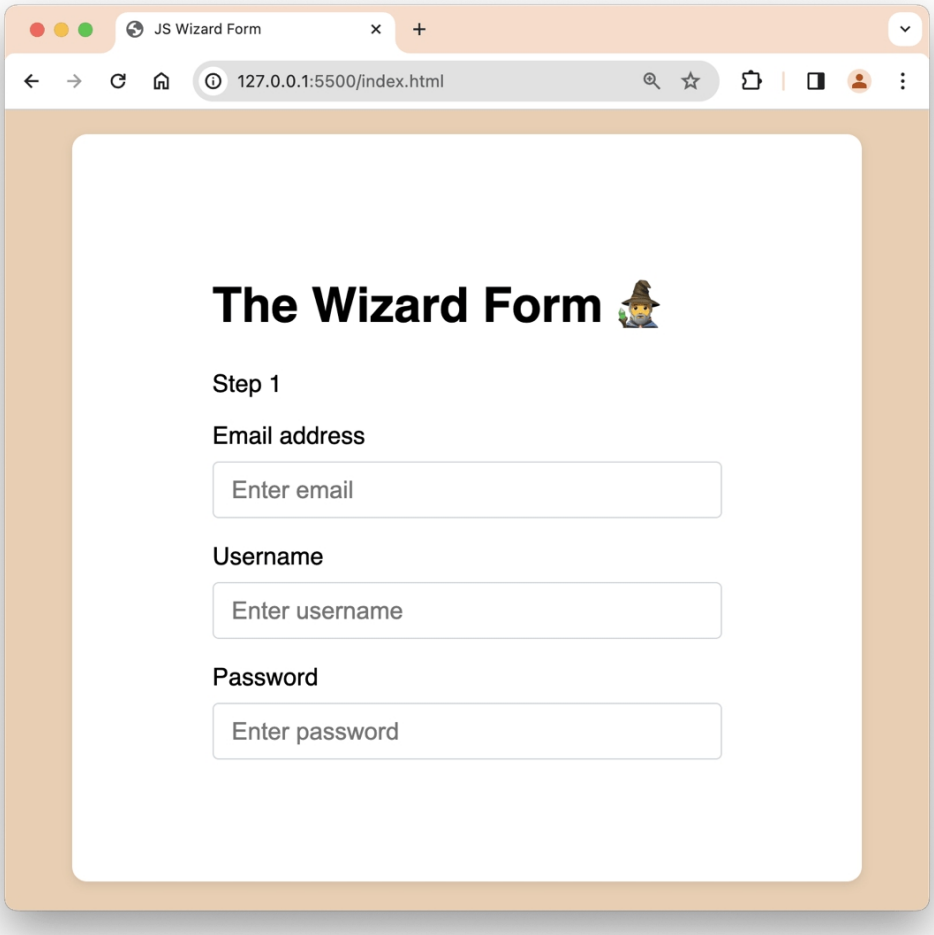
I'll briefly explain these rules below:

```
body {
  margin: 1rem;
  font-family: sans-serif;
  background-color: #edceb0;
}

.container {
  margin-left: auto;
  margin-right: auto;
  background: #fff;
  height: 424px;
  width: 338px;
  padding: 71px 93px 0;
  border-radius: 10px;
  box-shadow: 0 2px 7px 0 rgba(0, 0, 0, 0.1);
}

label {
  display: inline-block;
  margin-bottom: 0.5rem;
}

.form-group {
  margin-bottom: 1rem;
}

.form-control {
  width: 100%;
  padding: 0.375rem 0.75rem;
  font-size: 1rem;
  line-height: 1.5;
  color: #495057;
  border: 1px solid #ced4da;
  border-radius: 0.25rem;
  box-sizing: border-box;
}
```

The body style is used to add space, font family, background to the application. The `.container` style is used to beautify the div

containing the form. The `label`, `.form-group` and `.form-control` is used to make the form look good.

You can see the style applied in the browser:



The first step is done. We've added the minimum amount of HTML and CSS required for the document.

Summary

In this chapter, we laid the foundations for our Wizard Form Application by writing the HTML and CSS content.

The `id` and `class` attributes are added to the HTML elements to help in styling and manipulating with JavaScript in the next chapter.

If you need to check your work, you can get the sample code used in this chapter at https://github.com/nathansebhastian/js-wizard-form/tree/main/check-point-1

CHAPTER 11: THE WIZARD FORM MECHANISM

In this chapter, we're going to implement the Wizard Form mechanism using JavaScript. To do so, we're going to use the query selector methods provided by the DOM.

DOM Query Selector Methods

The Document Object Model provides a set of methods that allow you to select a specific element in the HTML nodes. They are the `querySelector()` and `querySelectorAll()` methods.

The `querySelector()` method is a JavaScript method from the `document` object that allows you to retrieve a single `Element` object that matches the CSS selector passed to the method.

The CSS selector is the pattern you use when targeting a specific element for styling. Here's an example of CSS selectors:

```
/* Selecting all elements with the id value of box */
#box {
}

/* Selecting all elements with the class value of btn */
```

```
.btn {
}
```

The pattern you need to pass to the `querySelector()` method can be as simple or as complex as you need.

For example, here's how to use the method to retrieve an element with an `id` attribute of 'box':

```
document.querySelector('#box');
```

The `querySelector()` method always returns the first element that matches the query. The method searches for the element from the top to the bottom of the DOM tree.

The string parameter passed to the `querySelector()` method follows the CSS selector pattern, where `class` is represented by a period `.` and `id` is represented by a hash `#`.

Suppose you have the following HTML element on your page:

```
<body>
  <h1>Query Selector</h1>
  <h2>Learning Query selector method</h2>
  <p id="opening" class="bold">Placeholder</p>
</body>
```

You can retrieve the <p> element by passing "p" as the method's argument:

```
let element = document.querySelector('p');

console.log(element);
// <p id="opening" class="bold">Opening</p>
```

Or you can also pass the `id` or the `class` attribute, it will return the same <p> element:

```
document.querySelector('.bold');
// <p id="opening" class="bold">Opening</p>

document.querySelector('#opening');
// <p id="opening" class="bold">Opening</p>
```

When you pass a selector that returns two elements, only the first element will be returned by the method. The following code tries to fetch both <h1> and <h2> elements:

```
document.querySelector('h1, h2');
// <h1>Query Selector</h1>
```

Above, you can see that the method returns the first match, which is the <h1> element.

And that's how the `querySelector()` method works. When you need to retrieve more than a single element, you need to use the alternative `querySelectorAll()` method instead.

querySelectorAll() Method

The `querySelectorAll()` method is a JavaScript method from the DOM API that allows you to retrieve all elements that match the CSS selector you passed as the argument.

For example, here's how to use the method to retrieve any element that has the `class` attribute of `box`:

```
document.querySelectorAll('.box');
```

The string parameter passed to the `querySelectorAll()` method follows the CSS selector pattern, where `class` is represented by a period `.` and `id` is represented by a hash `#`.

To retrieve all copies of a specific element, you can simply pass the name of the element as its argument.

Suppose you have the following HTML element on your page:

```
<body>
  <p id="opening" class="bold">Opening</p>
  <p id="middle">Middle</p>
  <p id="closing" class="bold">Closing</p>
</body>
```

You can retrieve all <p> elements by passing `"p"` as the method's argument:

```
let elements = document.querySelectorAll('p');

console.log(elements);
// NodeList(3) [p#opening.bold, p#middle, p#closing.bold]
console.log(elements[0]);
// <p id="opening" class="bold">Opening</p>
```

Or when you only need the opening and the closing <p> elements, you can pass either the class or the îdś as the argument:

```
document.querySelectorAll('.bold');
// [p#opening.bold, p#closing.bold]

document.querySelectorAll('#opening, #closing');
// [p#opening.bold, p#closing.bold]
```

You can also select elements by other attributes like `target` or `value`:

```
// return all elements with target="_blank"
document.querySelectorAll('[target=_blank]');

// return all elements with value="red"
document.querySelectorAll('[value=red]');
```

The return value of the `querySelectorAll()` method will be an array-like object called `NodeList`.

To access elements of the `NodeList` object, you can use the index position. You can also use the `forEach()` method to iterate over the elements:

```
let elements = document.querySelectorAll('p');

// 1. Access element at index 0
console.log(element[0]);

// 2. Iterate over the elements using forEach()
elements.forEach(function (element) {
  console.log(element);
  element.style.backgroundColor = 'yellow';
});
```

The code above will log the current element inside the loop and set its `background-color` value to `"yellow"`;

The `querySelectorAll()` method always returns a `NodeList` object even when you only have one matching element.

Hide Form Steps With CSS

Back to the Wizard Form Application, we need a way to show and hide the form steps using CSS and JavaScript. Add the following rules in `style.css`:

```
.step {
  display: none;
}

.step.active {
  display: block;
}
```

This causes all the div tags with the `step` class to be hidden from display. To show the form step add the class `active` to the `step1` div as follows:

```
<div id="step1" class="step active">
  <div class="form-group">...</div>
</div>
```

Go back to the browser, and you now see only the first step of the wizard form being displayed.

Adding the Next Button

Now we need some buttons to move between the form steps, so add the following button on `step1` div:

```
<div id="step1" class="step active">
  <div class="form-group">...</div>
  <button
    type="button"
    class="btn btn-primary f-right"
    onclick="nextStep()"
  >
    Next
  </button>
</div>
```

In this button, we add the `onclick` attribute so that we can run JavaScript code when the button is clicked.

The name of the function we want to call is `nextStep()`, so let's create that function in our `script.js` file. We also need a `showStep()` function that will :

```
let currentStep = 1;

function nextStep() {
  if (currentStep < 3) {
    currentStep++;
    document.querySelectorAll('.step').forEach(function (element) {
      element.classList.remove('active');
    });

    document.querySelector(`#step${currentStep}`).classList.add('active');
    document.querySelector('#currentStep').textContent = currentStep;
  }
}
```

In the `script.js` file, we declared a variable named `currentStep`, which is used to let us know in what step the form is currently in.

In the `nextStep()` function, we used the `if` statement to check if the `currentStep` is less than 3 because the form only has 3 steps.

To move to the next step, we increment the `currentStep` value by one using the `++` operator, then we use the `document.querySelectorAll()` method to select all elements containing the `step` class. We remove the `active` class from all form steps.

After that, we select the current step and add the `active` class to the element using the `classList.add()` method. The final task is to change the content of the `currentStep` paragraph to the `currentStep`.

Open the wizard form in the browser, and you should be able to navigate to step 3 using the next button. But notice that you can't go back to the previous step! This is what we're going to implement next.

Adding the Previous Button

Now we need another button to move to the previous step. On `step2` and `step3` div tags, you can add the following buttons:

```
<div id='step2' class='step'>
  <div class='form-group'>...</div>
  <button type='button' class='btn btn-secondary' onclick='prevStep()'>
    Previous
  </button>
  <button type='button' class='btn btn-primary f-right' onclick='nextStep()'>
    Next
  </button>
</div>

<div id="step3" class="step">
  <div class="form-group">...</div>
  <button type="button" class="btn btn-secondary" onclick="prevStep()">
    Previous
  </button>
</div>
```

Next, we need to add a `prevStep()` function to the `script.js` file:

```
function prevStep() {
  if (currentStep > 1) {
    currentStep--;
    document.querySelectorAll('.step').forEach(function (element) {
      element.classList.remove('active');
    });

    document.querySelector(`#step${currentStep}`).classList.add('active');
    document.querySelector('#currentStep').textContent = currentStep;
  }
}
```

The `prevStep()` function is identical to the `nextStep()` function, except for the `if` statement check and decrementing the `currentStep` part.

Instead of writing the same code in two different functions, we can create a new function that runs the query selectors part. Let's name this function `showStep()`.

Your `script.js` file should look as follows:

```
let currentStep = 1;

function showStep() {
  document.querySelectorAll('.step').forEach(function (element) {
    element.classList.remove('active');
  });

  document.querySelector(`#step${currentStep}`).classList.add('active');
  document.querySelector('#currentStep').textContent = currentStep;
}

function nextStep() {
  if (currentStep < 3) {
    currentStep++;
    showStep();
  }
}

function prevStep() {
  if (currentStep > 1) {
    currentStep--;
    showStep();
  }
}
```

Back to the browser, now you can go to the next and previous steps with the buttons. The last thing we need to do is to style these buttons. Right now they look pretty bad:

```
.f-right {
  float: right !important;
}

.btn {
  color: #fff;
  cursor: pointer;
  font-size: 1rem;
  line-height: 1.5;
  border-radius: 0.25rem;
  padding: 0.375rem 0.75rem;
  border: 1px solid transparent;
}

.btn-next {
  border-color: #007bff;
  background-color: #007bff;
}

.btn-previous {
  border-color: #6c757d;
  background-color: #6c757d;
}
```

The `.f-right` style adds the `float` property to the element to move it to the right, the `.btn` is used to style the buttons.

The `.btn-next` and `.btn-previous` styles add border and background colors to the respective elements.

So far so good. The final task is to add the submit button and display the data collected by the form.

If you get an error and would like to check your work, you can get the code used in this chapter at https://github.com/nathansebhastian/js-wizard-form/tree/main/check-point-2

CHAPTER 12: HANDLING WIZARD FORM SUBMISSION

To finish the Wizard Form application, we need to provide instructions on what to do when the form is submitted.

Adding a Submit Button

Back to the project, the final task we need to do on the Wizard Form is to add a submit button to the third step, so let's do it.

Add the submit button to the `step3` div as follows:

```
<div id="step3" class="step">
  <div class="form-group">
    <label for="password">Password</label>
    <input
      class="form-control"
      name="password"
      type="password"
      placeholder="Enter password"
    />
  </div>
  <button type="button" class="btn btn-previous" onclick="prevStep()">
    Previous
  </button>
  <button type="submit" class="btn btn-submit f-right">Submit</button>
</div>
```

Add the CSS for the submit button in `style.css` as follows:

```
.btn-submit {
  border-color: #28a745;
  background-color: #28a745;
}
```

Now we need to listen when the submit button is clicked.

Listening to Form Submit Events Using JavaScript

Besides the DOM API, the browser also expose signals known as events that you can use as a cue to run a specific code. We have done this earlier when adding the `onclick` attribute to the buttons:

```
<button type='button' class='btn btn-secondary' onclick='prevStep()'>
  Previous
</button>
<button type='button' class='btn btn-primary f-right' onclick='nextStep()'>
  Next
</button>
```

The `onclick` HTML attribute is a shortcut to listen to the click event. When the previous button is clicked, we run the `prevStep()` function. When the next button is clicked, we run the `nextStep()` function.

Aside from using the `onclick` attribute, we can also add event listeners using the `addEventListener()` method.

Let's add an event listener so that we know when a form is submitted. The Wizard Form we created has an `id` attribute, so you can select that form using JavaScript.

```
const wizardForm = document.querySelector('#wizard-form');
```

Next, you need to call the `addEventListener()` method and listen to the 'submit' event as follows:

```
const wizardForm = document.querySelector('#wizard-form');

wizardForm.addEventListener('submit', function (event) {
  // TODO: handle the submit event
});
```

Now that we can listen to the submit event, we need to interrupt the default behavior of the form so that JavaScript can take over.

To do so, we need to call the `event.preventDefault()` method inside the function we passed to the event listener:

```
const wizardForm = document.querySelector('#wizard-form');

wizardForm.addEventListener('submit', function (event) {
  event.preventDefault();
});
```

The `preventDefault()` method interrupts the default flow of the submit event, which will cause the browser to refresh the current page.

From here, you can instruct JavaScript to do specific tasks to the form data submitted by the user.

The form object has an object property named `elements` that store all input elements you define inside the form. Inside the object, you can access the value of input fields by specifying its `name` attribute, followed by the `value` property.

Here's how to get the values form our Wizard Form:

```
event.preventDefault();
const email = wizardForm.elements.email.value;
const username = wizardForm.elements.username.value;
const password = wizardForm.elements.password.value;
```

Finally, we display the data using the JavaScript `alert()` function, which

```
const email = wizardForm.elements.email.value;
const username = wizardForm.elements.username.value;
const password = wizardForm.elements.password.value;

alert(`Registration complete! Your details: \n
  Email: ${email} \n
  Username: ${username} \n
  Password: ${password}`);
```

You can test the Wizard Form in the browser. Fill in the textboxes and you'll get an alert when clicking the submit button:

Just to make the application smarter, we can reset the form after it has been submitted by the user:

```
  const email = wizardForm.elements.email.value;
   const username = wizardForm.elements.username.value;
   const password = wizardForm.elements.password.value;

//   alert(...);

   wizardForm.reset();
   currentStep = 1;
   showStep();
```

The `reset()` method from the form will reset all textboxes, then we set the `currentStep` variable to 1 and run the `showStep()` function to reset the interface.

Taking Care of Repeating Code

If you look closely, you might notice that the code for the Next and Previous buttons in the steps are identical. Instead of writing them in the HTML file, we can use JavaScript to add them dynamically.

First, select all the steps div using `querySelector()` as follows:

```
const stepsDiv = document.querySelectorAll('.step');

stepsDiv.forEach((element, index) => {
  if (index === 0) {
    element.innerHTML += nextButton;
  } else if (index === stepsDiv.length - 1) {
    element.innerHTML += previousButton;
    element.innerHTML += registerButton;
  } else {
    element.innerHTML += previousButton;
    element.innerHTML += nextButton;
  }
});
```

Next, declare three variables containing the HTML buttons tag:

```
const stepsDiv = document.querySelectorAll('.step');

const previousButton =
  '<button class="btn btn-secondary" onclick="prevStep()"
type="button">Previous</button>';

const nextButton =
  '<button class="btn btn-primary f-right" onclick="nextStep()"
type="button">Next</button>';

const registerButton =
  '<button type="submit" class="btn btn-success f-right">Register</button>';
```

Now, we need to iterate over the elements using the `forEach()` method. We can check if the element is the first or the last step from the `index` value.

For the first element at index `0`, we only add the next button. For the last element, we add the previous and submit buttons, for all the middle elements, we add the previous and next buttons:

```
stepsDiv.forEach((element, index) => {
  if (index === 0) {
    element.innerHTML += nextButton;
  } else if (index === stepsDiv.length - 1) {
    element.innerHTML += previousButton;
    element.innerHTML += registerButton;
  } else {
    element.innerHTML += previousButton;
    element.innerHTML += nextButton;
  }
});
```

To check if the element is the last step of the Wizard Form, we compare the `index` value with the `stepsDiv.length - 1` value.

To add an element to existing elements, we use the `+=` assignment operator.

Now we can remove all `<button>` elements from the HTML file. Run the Wizard Form in the browser again, and you'll see the buttons added using JavaScript.

The Wizard Form is finished. You can check the completed application at https://github.com/nathansebhastian/js-wizard-form/tree/main/complete

Summary

Congratulations on finishing your first JavaScript project! The Wizard Form application helps you learn the idea of using JavaScript to manipulate what you see on the screen.

Although a Wizard Form looks complex, it's actually pretty easy when you have the steps mechanism created using JavaScript.

Let's recap what we've learned in this project:

1. How to select HTML elements using `document.querySelector()` and `document.querySelectorAll()`
2. Show and hide elements using CSS and JavaScript
3. Conditionally display HTML elements using variables, `if` statements, and functions
4. An introduction to browser events
5. Handling a form submit event with `addEventListener()`
6. Retrieve, display, and reset form data with the `event` object
7. Add elements to specific parts of the HTML document with the `innerHTML` property

And the most important thing is that you've seen how an application can be built step-by-step and gradually becomes more complex.

In the next chapter, we're going to learn how to use JavaScript to send and receive data from a backend service. See you there.

CHAPTER 13: ASYNCHRONOUS JAVASCRIPT AND CALLBACKS

JavaScript is a non-blocking, synchronous programming language.

What does this even mean? It means that JavaScript doesn't wait until a code finished running before running the next one (non-blocking)

Synchronous means sequential, and we know this because JavaScript runs code line by line from the top to the bottom.

Why is this important? Let me show you an easy example.

The `window` object of the DOM has a method called `setTimeout()` that you can use to run a function after a specific amount of time.

The method accepts two arguments:

1. The function to execute after the timeout
2. The amount of time to wait in milliseconds

For example, here's how to wait 3 seconds before running a console log:

```
setTimeout(function () {
  console.log('Hello!');
}, 3000);
```

Run the code above from the console, and after 3 seconds, you should see the 'Hello!' string printed.

Now, let's add some console log before and after the `setTimeout()` method as follows:

```
console.log('Start');
setTimeout(function () {
  console.log('Hello!');
}, 3000);
console.log('End');
```

Now we have 'Start' and 'End' strings which should be printed before and after the `setTimeout()` method has finished executing.

When you run the code above, you will see this output:

image::images/3-4-example-settimeout.png

Surprise! The 'End' string got printed ahead of the 'Hello!' string. This is because of the non-blocking nature of JavaScript.

The synchronous nature of JavaScript makes it read the code line by line from top to bottom. But because it's non-blocking, JavaScript executes the `setTimeout()` method, then continue code execution to the next line.

This is not a bug. It's an expected behavior of JavaScript because of the way it's designed.

And this is important to know because when you're developing a website or application, there will be times when you send data to a server, and you need to wait for the response before continuing code execution.

To make JavaScript wait until a specific operation has been completed, we need to use what's known as the callback function.

Callback Functions Explained

A callback function is simply a function passed into another function as an argument. The callback function would then be used by the function when the situation is right.

The function we passed into the `setTimeout()` method above is a callback function. So does the function we passed into the `addEventListener()` method.

Back to the example above, we can make JavaScript wait for the `setTimeout()` method to finish before printing the 'End' string by moving the console log line below 'Hello!' as follows:

```
console.log('Start');

setTimeout(function () {
  console.log('Hello!');
  console.log('End!');
}, 3000);
```

But of course, we can do this because we're the ones who added the `setTimeout()` method.

If we imagine the `setTimeout()` method as a part of code that we can't touch, like the `addEventListener()` method, how do we make sure that it runs a certain instruction after the timeout?

The answer is to use a callback, just like the `addEventListener()` method. In the example below, imagine that you can't modify the content of the `processData()` function:

```
function processData(data, callbackFn) {
  setTimeout(function () {
    console.log('Sending data back');
    const response = {
      data: data + 'abc',
    };

    callbackFn(response);
  }, 3000);
}
```

Here, the `processData()` function accepts two parameters:

1. The `data` parameter to process
2. The callback function to execute after the process

To get the `response` from the function above, we need to call that function and pass both data and a callback function.

Here's how you do it:

```
processData('Nathan', function (response) {
  console.log(`Response given: ${response}`);
});
```

This callback pattern is used not only when responding to events, but also when sending data to another server.

We don’t know in advance how long a server needs to process the data, so we put a callback function that will be executed when the response is ready.

Summary

In this chapter, you’ve learned that JavaScript is a programming language that doesn’t stop to wait for a function that takes time to finish. It will continue to run the next line of code when the previous line isn’t completed yet.

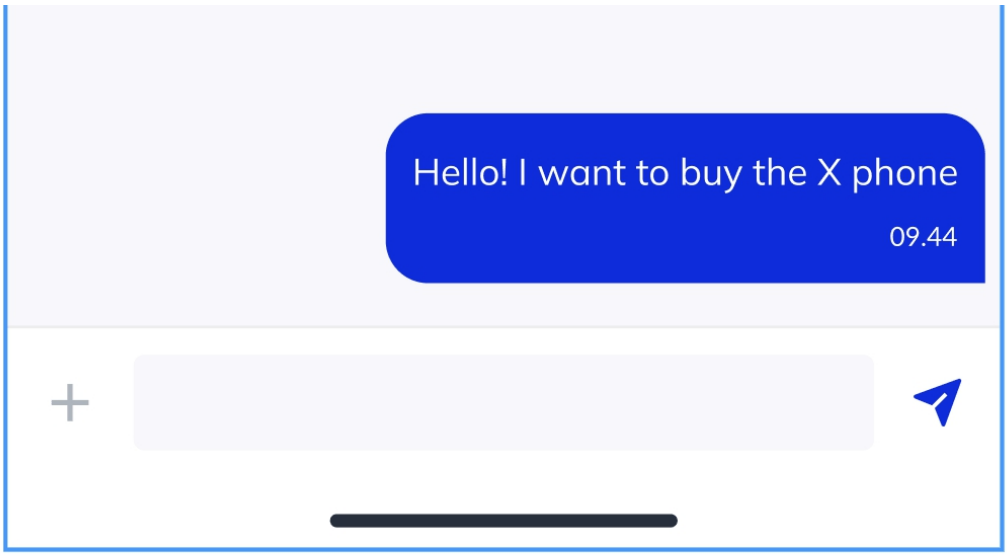
The callback function is a way to make JavaScript run in an asynchronous manner. The next step of the operation starts only when the previous operation is completed.

CHAPTER 14: PROMISES

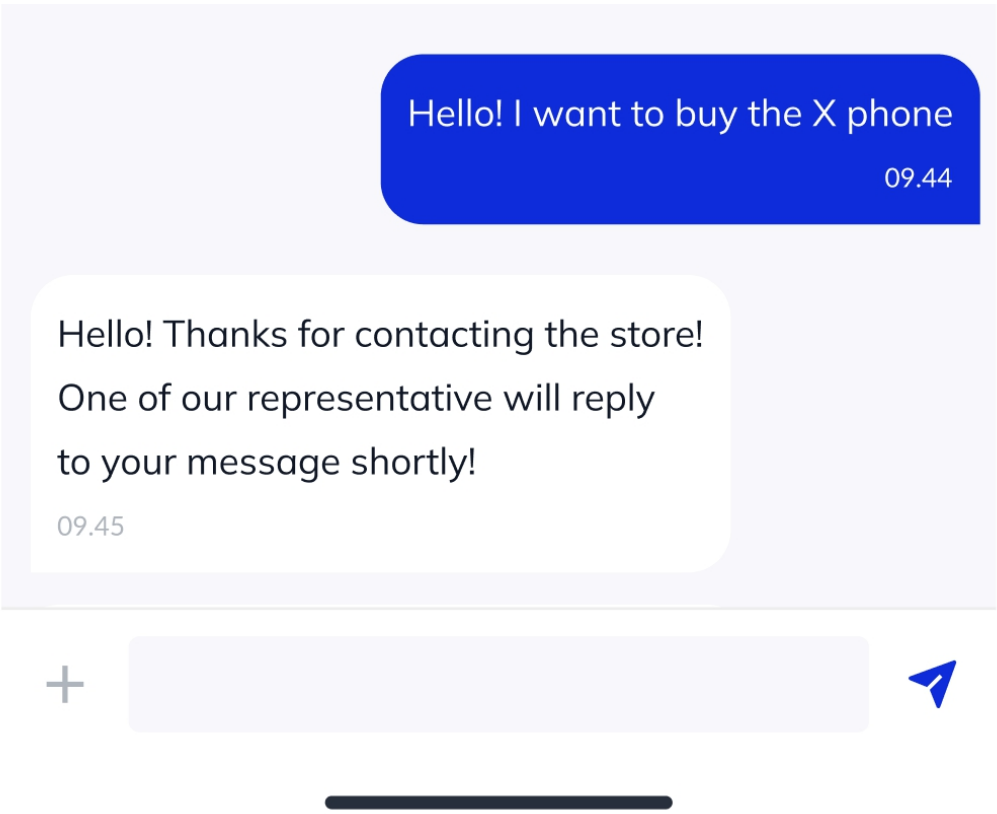
A Promise is an alternative way to deal with the non-blocking nature of JavaScript while running asynchronous operations. It's a modern feature released in 2015 (that is 20 years after the first release of JavaScript)

Basically, a `Promise` object represents a "pending state" in the most common sense: The promise will eventually be fulfilled at a later date.

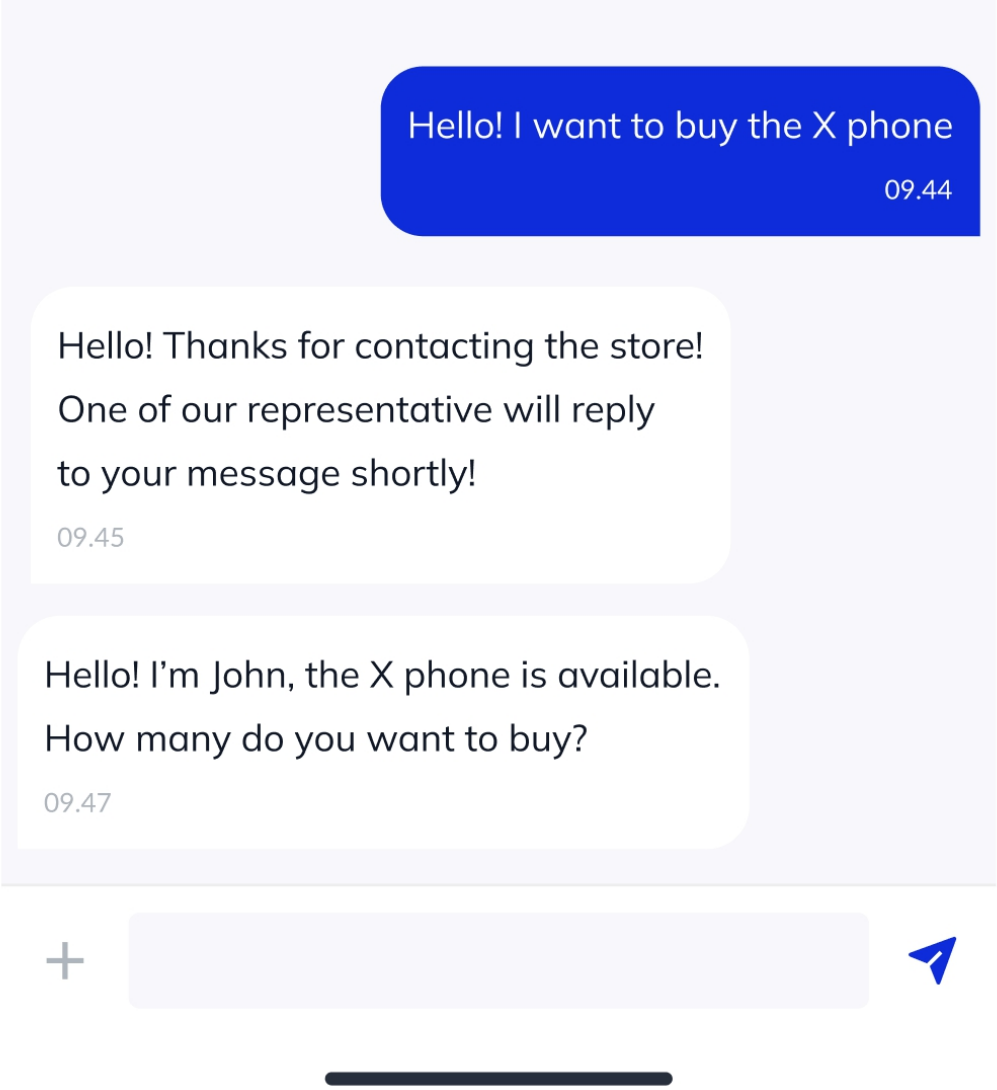
To give you an illustration, suppose you want to buy a new phone to replace your old phone, so you open a messaging app to contact a phone store. This is similar to how you access a variable or a function that returns a promise:



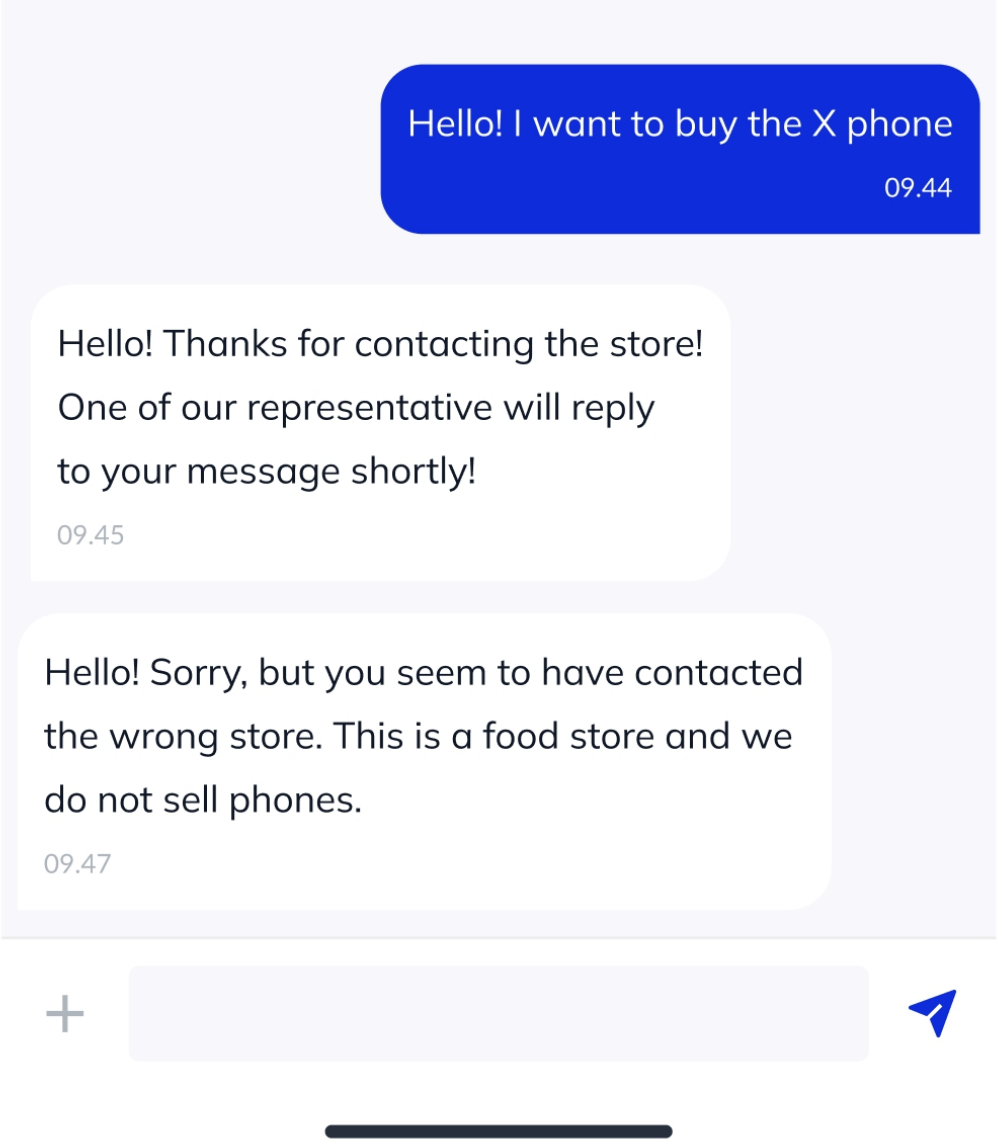
After you send a message explaining what you want, you get an automated message saying that a representative will answer your message shortly. This is similar to receiving a Promise object:



A minute later, you get a new message from a human representative, saying that the phone model you want to buy is available for purchase. This is when the Promise was resolved:



Or, in a completely different scenario, the representative tells you that the store doesn't sell phones, because the store is a food store and not a phone store. This means the Promise was rejected:



This illustration shows how the `Promise` object in JavaScript works:

- A Promise is like the automated message that we've seen earlier, it represents a pending state that must be fulfilled by later.
- The human representative saying that the phone model is available is similar to the `resolve()` method, which shows

that the Promise is fulfilled.

- The representative telling you that you're contacting the wrong store is like the `reject()` method, which is the method used to show that the Promise can't be fulfilled because of an error.

A typical promise implementation would look as follows:

```
let p = new Promise((resolve, reject) => {
  let isTrue = true;
  if (isTrue) {
    resolve('Promise resolved');
  } else {
    reject('Promise rejected');
  }
});
```

When creating a new Promise object, we need to pass a callback function that will be called immediately with two arguments: the `resolve()` and `reject()` functions

Depending on the result of the `Promise`, either the `resolve()` or the `reject()` function will be called to end the pending state.

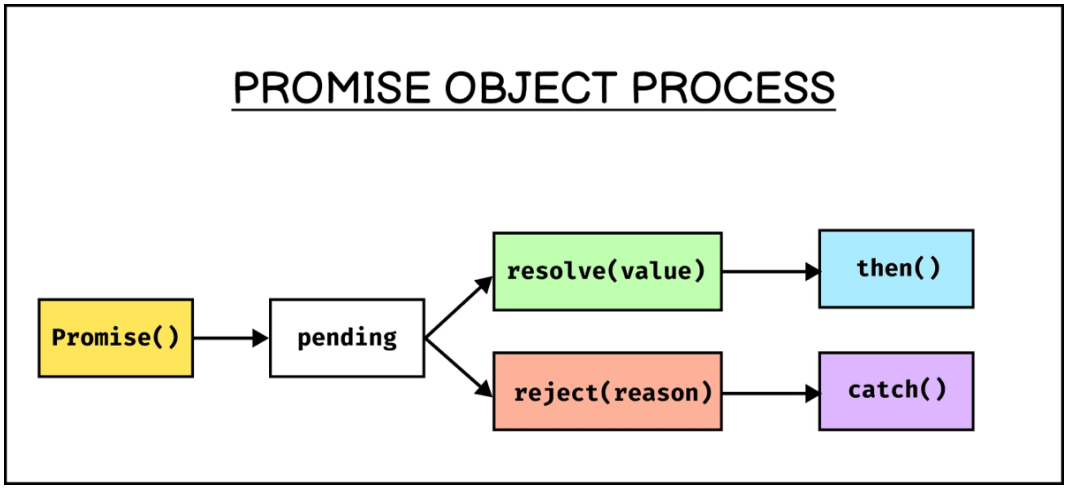
To handle the `Promise` object, you need to chain the function call with `then()` and `catch()` functions as shown below:

```
let p = new Promise((resolve, reject) => {
  let isTrue = true;
  if (isTrue) {
    resolve('Success');
  } else {
    reject('Error');
  }
});

p
```

```
.then(message => console.log(`Promise resolved: ${message}`))
.catch(message => console.log(`Promise rejected: ${message}`));
```

The `resolve()` function corresponds to the `then()` function, while `reject()` corresponds to the `catch()` function. You can change the `isTrue` value to `false` to test this.

Here's an illustration of the promise process:



Using the promise pattern, you can call your functions sequentially by placing the next process inside the `then()` method.

Callbacks vs Promises

The promise pattern is created to replace the use of callbacks. By using promises, the code we write would be more intuitive and maintainable.

Going back to the messaging illustration, let's create an example of using callbacks to handle the situation. First, we declare the

two variables required for this situation, called `isPhoneStore` and `isPhoneAvailable`:

```
const isPhoneStore = true;
const isPhoneAvailable = true;
```

Next, we write a function that will process incoming messages. This function will mimic the promise pattern, and it will resolve only when `isPhoneStore` and `isPhoneAvailable` are `true`:

```
function processMessage(resolveCallback, rejectCallback) {
  if (!isPhoneStore) {
    rejectCallback({
      name: 'Wrong store',
      message: 'Sorry, this is a food store!',
    });
  } else if (!isPhoneAvailable) {
    rejectCallback({
      name: 'Out of stock',
      message: 'Sorry, the X phone is out of stock!',
    });
  } else {
    resolveCallback({
      name: 'OK',
      message: 'The X phone is available! How many you want to buy?',
    });
  }
}
```

Here, you can see that the function `processMessage` accepts two callback functions: `resolveCallback` and `rejectCallback`.

When we call the function, we need to provide the callback functions, similar to how we need to chain the `then()` and `catch()` methods when accessing a promise:

```
processMessage(
  value => console.log(value),
  reason => console.log(reason)
);
```

In the call to the `processMessage` above, the first argument is the `resolveCallback()` function, and the second argument is the `rejectCallback()` function.

If you run the code above, then the `resolveCallback()` function will be called. You can change one of the two variables to `false` to trigger the `rejectCallback()` function.

Now that we have a working callback example, let's rewrite the code using a promise as follows:

```
const isPhoneStore = true;
const isPhoneAvailable = true;

function processMessage() {
  return new Promise((resolve, reject) => {
    if (!isPhoneStore) {
      reject({
        name: 'Wrong store',
        message: 'Sorry, this is a food store!',
      });
    } else if (!isPhoneAvailable) {
      reject({
        name: 'Out of stock',
        message: 'Sorry, the X phone is out of stock!',
      });
    } else {
      resolve({
        name: 'OK',
        message: 'The X phone is available! How many you want to buy?',
      });
    }
  });
}

processMessage()
  .then(response => console.log(response))
  .catch(error => console.log(error));
```

Here, you can see that the `processMessage()` function returns a `Promise` object that gets resolved only when both `isPhoneStore` and `isPhoneAvailable` are `true`.

When one of the two variables is `false`, then the `Promise` object will be rejected.

Here you can see that you don't need to add two extra parameters to the `processMessage()` function just for the callbacks, and when calling the function, you also use the `then()` and `catch()` methods to handle the result of the promise.

The use of a promise makes the code easier to understand. Here's the comparison of the two side by side:

Summary

In this chapter, you've learned how the `Promise` object works in JavaScript. A promise is easy to understand when you realize

the three states that can be generated by the promise: pending, resolved, and rejected.

When facing a promise, you need to define what's going to happen next inside the `.then()` and `.catch()` methods

You've also learned how promises can be used to replace callbacks.

CHAPTER 15: THE FETCH API

In this chapter, we're going to learn about the Fetch API. But before we start, let's quickly review what we mean by API.

What is an API?

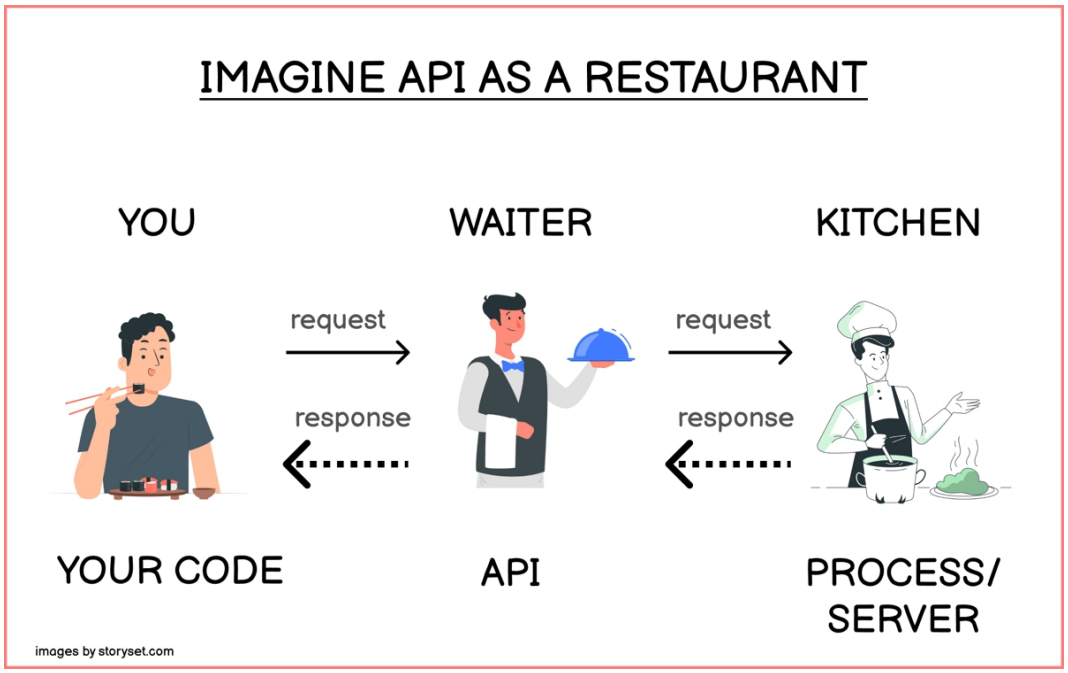
API stands for Application Programming Interface, but this fancy term doesn't really tell you what an API is, so let me explain it in simpler terms.

An API is simply a program that's exposed by the developer. The intention of exposing this program is so that other applications can call and use the program to achieve a specific result.

For example, imagine you're going to a restaurant to have lunch. You don't directly go to the kitchen and prepare the food.

Instead, you order food from the waiter, who takes the order to the kitchen. In the kitchen, the chef will make the food, and when it's done, the waiter brings the food to your table.

You know nothing about what's happening in the kitchen. All you know is that you specify what you want to the waiter, and the waiter takes care of the rest.



In a similar way, an API serve as the gateway that you can access to achieve a specific result. You don't really know how the API works under the hood. All you know is that you get what you wanted from the API.

There are two kinds of APIs: the web API (also known as REST API), or library API.

The web API is created by exposing certain endpoints that you can access to send and receive data.

The library API is a piece of code written by other people. This code usually has a bunch of objects, methods, and properties that you can access to do something. You don't need to know how they work under the hood.

The `document` object exposed by the browser is an example of a library API, also known as the DOM API. You don't write the `document` object from scratch, and when you call the

`querySelector()` method, you don't know exactly what code is written in that method.

The Fetch API we're going to learn next is also a library API.

How Fetch API works

The Fetch API is a function that you can use to send a request to any web API URL and get some response.

To send a GET request similar to that of an HTML form, you only need to pass the URL where you want to send the data as an argument to the `fetch()` function:

```
fetch('<Your URL>');
```

The `fetch()` function accepts two parameters:

1. The URL to send the request to
2. The options to set in the request. You can set the request method to GET or POST here.

Under the hood, the `fetch()` function returns a promise, so you need to add the `.then()` and `.catch()` methods.

When the request returns a response, the `then()` method will be called. If the request returns an error, then the `catch()` method will be executed.

Inside the `.then()` and `.catch()` methods, you pass a callback function to execute when the respective methods are called. In

the following example, we're going to hit a dummy URL located in jsonplaceholder.typicode.com:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => console.log(response))
  .catch(error => console.log(error));
```

The code above will give the following response:

Here, you can see that the body property contains a `ReadableStream`. To use the `ReadableStream` in our JavaScript application, we need to convert it to JSON format first, so let's do it:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(data => console.log(data))
```

The `json()` method converts the `ReadableStream` into a JavaScript object. The `data` variable above will be printed as follows:

```
{
  userId: 1,
  id: 1,
  title: "delectus aut autem",
  completed: false
}
```

If you want to send a POST request instead of a GET request, pass an object containing a `method` property as follows:

```
fetch('https://jsonplaceholder.typicode.com/todos', {
  method: 'POST',
}).then(response => response.json())
  .then(data => console.log(data))
```

When you send a POST method, you need to set the request header and body properties to ensure a smooth process.

For the header, you need to add the `Content-Type` property and set it to `application/json`. The data you want to send should be put inside the body property in a JSON format. See the example below:

```
fetch('https://jsonplaceholder.typicode.com/todos', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ title: 'Learn JavaScript' }),
}).then(response => response.json())
  .then(data => console.log(data))
```

In the example above, we send a POST request to create a new todo task. The title of the task is Learn JavaScript. The response from the typicode API would be similar as follows:

```
{
  title: 'Learn JavaScript',
  id: 201
}
```

Summary

The Fetch API allows you to access APIs and perform a network request using standard request methods such as GET, POST, PUT, PATCH, and DELETE.

The Fetch API returns a promise, so you need to chain the function call with `.then()` and `.catch()` methods, as shown in the previous chapter.

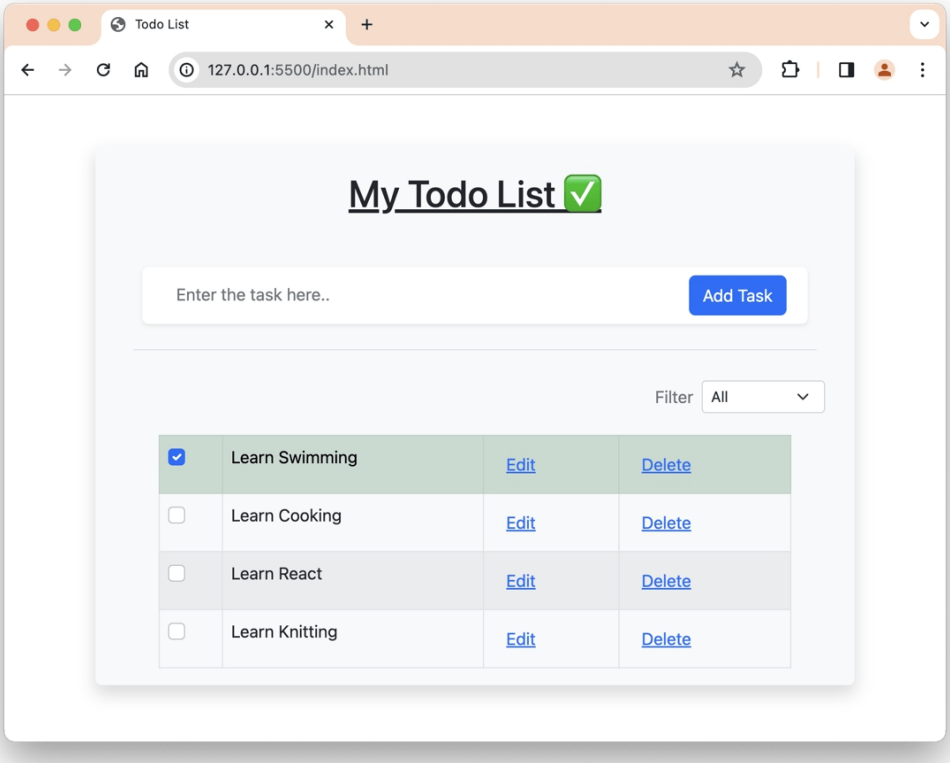
Here, we only see the GET and POST methods in action, but don’t worry because we’re going to use the rest in the following project.

CHAPTER 16: PROJECT 2 - CREATING A TO-DO LIST APPLICATION

As the final project of this book, we're going to create a To-do List application.

In this application, we can add, edit, delete, and mark a task as completed.

The finished application looks as follows:



To start building this application, we need to create a simple server that's going to store our task data. To do so, we need to install Node.js

Installing Node.js

Node.js is a JavaScript runtime application that enables you to run JavaScript outside of the browser. We need to install this application on our computer to create a simple server.

You can download and install Node.js from https://nodejs.org. Pick the recommended LTS version because it has long-term support.

Creating the Server and Database

To set up this project, let's create a new folder named *todo-list*. Open this folder in VSCode, then create a new file named *db.json* and put the following content in it:

```
{
  "tasks": [
    {
      "title": "Learn Swimming",
      "completed": true,
      "id": 1
    },
    {
      "title": "Learn Cooking",
      "completed": false,
      "id": 2
    }
  ]
}
```

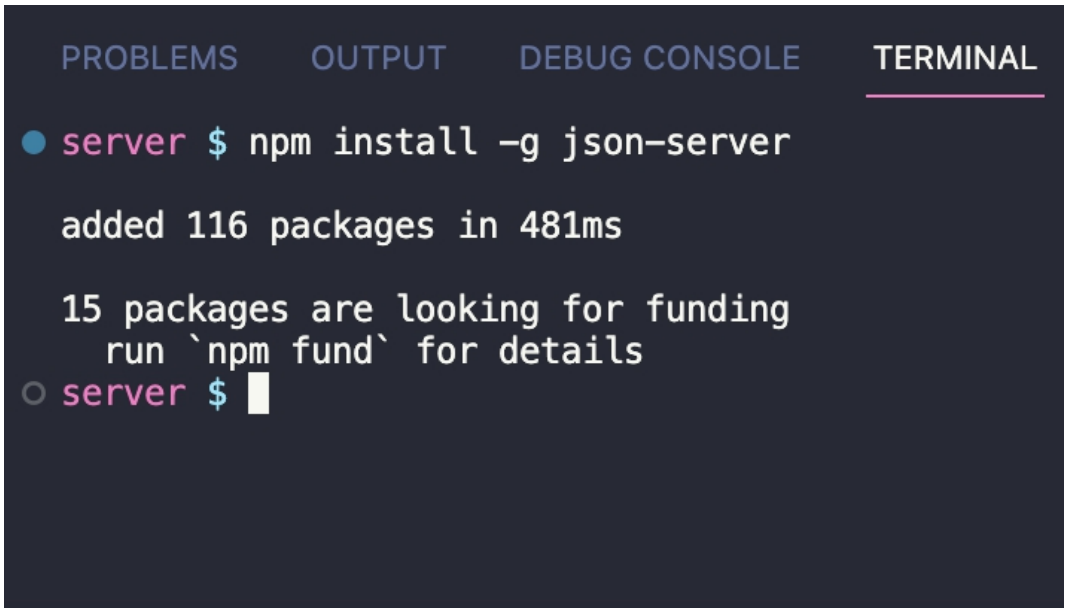
This JSON file will serve as the database of our application. When we add a new task, this file will be updated.

The next step is to run this JSON file as a server. To do so, you need to use Node Package Manager (or NPM for short) to install a json server.

First, right-click on the *todo-list* folder and select the Open in Integrated Terminal option. This will open the terminal from which you can run some commands.

The command I want you to run is `npm install -g json-server`. This command will install the `json-server` package globally on your computer.

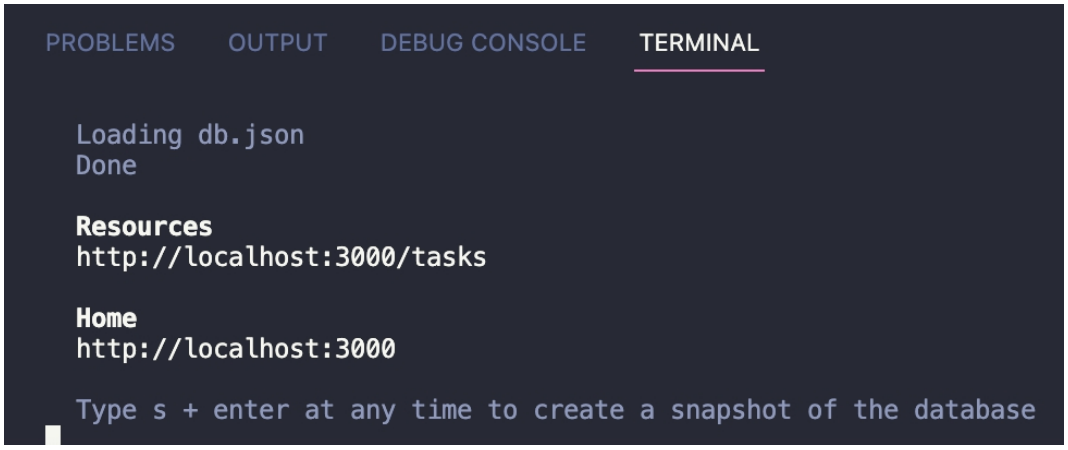
Once the installation is finished, you should see the following output:



The next step is to run the `json-server` package. Run this command in your terminal:

```
json-server db.json
```

If you see the following output, it means the JSON server is running successfully:



You can open the browser and access the URL at http://localhost:3000/tasks to see the JSON data we created earlier.

The To-do List application we're going to create is going to access this API endpoint for fetching, adding, removing, and updating the tasks.

You've created a simple server that exposes an API endpoint successfully. Nice work!

Let's Start the Project: HTML and CSS First

The next step is to write the front-end part of the project. Open the *client* folder in VSCode, then create two files named `index.html` and `script.js`.

Open the `index.html` file and write the standard HTML document structure just like when we create the Wizard Form:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>To-do List</title>
  </head>
  <body></body>
</html>
```

To focus our attention on the JavaScript part, we're going to use Bootstrap to style our HTML tags. Bootstrap is a library containing pre-written CSS rules that you can apply to your project.

To use Bootstrap styles, you just add specific classes to your HTML elements. Add Bootstrap to our project by creating a link that fetches the Bootstrap CSS from the CDN server as follows:

```
<title>To-do List</title>
<link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/css/bootstrap.min.css
"
/>
```

CDN stands for Content Delivery Network, and it's a server that allows you to fetch popular libraries. By using a CDN, you don't need to download the Bootstrap CSS file onto your computer and load it locally.

The last step is to add the `script.js` file to the HTML file. As always, add the `defer` attribute so that the browser loads the script after the HTML document has been rendered:

```
<script src="script.js" defer></script>
</head>
```

Now we need to add some HTML to the <body> tag. First, write the container that's going to contain all our HTML content. I'm going to add some comments to help you put the HTML in the right place:

```
<body>
  <div class="container m-5 rounded mx-auto bg-light shadow">
    <!-- App title section START-->
    <div class="row p-4">
      <div class="col">
        <h1 class="text-center">
          <u>My Todo List □</u>
        </h1>
      </div>
    </div>
    <!-- App title section END -->
  </div>
</body>
```

Under the 'App title section END' comment, add an input and a button we're going to use to create a task. The divs around the input and the button is created only to apply Bootstrap style:

```
<!-- Create task section START -->
<div class="row p-3">
  <div class="col col-11 mx-auto">
    <div class="row bg-white rounded shadow-sm p-2">
      <div class="col">
        <input
          type="text"
          id="task-title"
          class="form-control border-0 rounded"
          placeholder="Enter the task here.."
        />
      </div>
      <div class="col-auto">
        <button type="button" class="btn btn-primary" onClick="addTask()">
          Add Task
        </button>
      </div>
    </div>
  </div>
</div>
<div class="my-2 mx-4 border-bottom"></div>
<!-- Create task section END -->
```

Next, we're going to add the filter section under the 'Create task section'. This filter section will contain a select input:

```
<!-- Filter options section START -->
<div class="row p-3 justify-content-end">
  <div class="col-auto d-flex align-items-center">
    <label class="text-secondary my-2 me-2">Filter</label>
    <select class="form-select form-select-sm" onChange="filterTasks(event)">
      <option value="all" selected>All</option>
      <option value="completed">Completed</option>
      <option value="active">Active</option>
    </select>
  </div>
</div>
<!-- Filter options section END -->
```

The last step is to display the tasks. We're going to use table element here:

```
<!-- Tasks list section START -->
<div class="row px-5">
  <div class="col">
    <div class="table-responsive">
      <table class="table table-striped table-bordered table-hover">
        <tbody id="tbody-tasks">
          <tr>
            <td><input class="form-check-input" type="checkbox" checked />
</td>
            <td>Learn JavaScript</td>
            <td>
              <button type="button" class="btn btn-link">Edit</button>
            </td>
            <td>
              <button type="button" class="btn btn-link">Delete</button>
            </td>
          </tr>
          <tr>
            <td><input class="form-check-input" type="checkbox" /></td>
            <td>Learn HTML</td>
            <td>
              <button type="button" class="btn btn-link">Edit</button>
            </td>
            <td>
              <button type="button" class="btn btn-link">Delete</button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
<!-- Tasks list section END -->
```

And now the HTML part is completed. Right-click on the `index.html` file and select Open with Live Server to see the result.

If you need to check your code, you can get the code used in this chapter at https://github.com/nathansebhastian/js-todo-list/tree/main/check-point-1

Summary

In this chapter, we've installed Node.js and created a simple server using the `json-server` package. The database we use in this project is a simple JSON file named `db.json`.

We've also created the HTML file for our To-do application and used Bootstrap to style the elements.

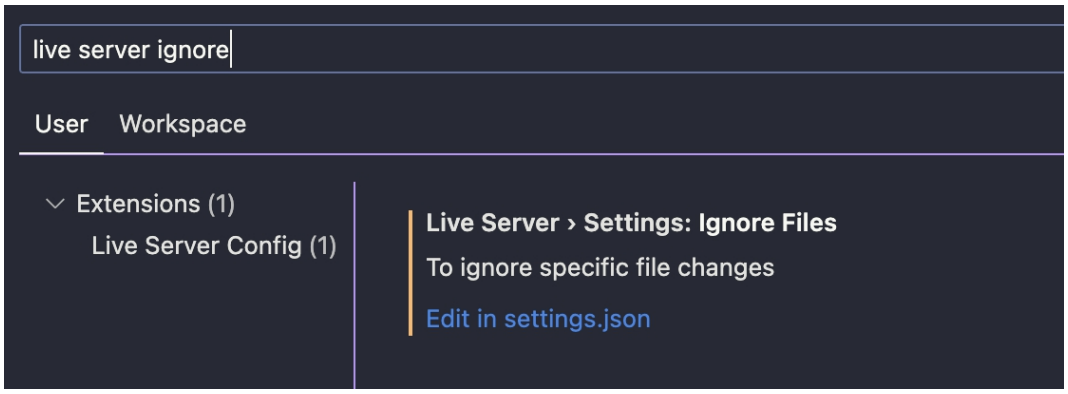
CHAPTER 17: ADDING CRUD FUNCTIONALITIES TO THE APPLICATION

Create, Read, Update, and Delete (CRUD) are four basic functionalities that make the core of a web application. We're going to learn how to implement them using JavaScript in this chapter.

Before we start creating the functionalities for the To-do List, we need to edit the Live Server settings and prevent server reload when the `db.json` file changes.

If we don't do this, the application will be interrupted when we add, edit, or delete a task.

Open the VSCode Settings menu by pressing `Ctrl + ,` or `Command + ,` and search for 'live server ignore'. You should see the following output:



Click the Edit in settings.json link, and add `db.json` to the Live Server ignore files settings as shown below:

```
"liveServer.settings.ignoreFiles": [
    "db.json",
    ".vscode/**",
    "**/*.scss",
    "**/*.sass",
    "**/*.ts"
]
```

Save the changes. Now Live Server won't reload when there's a change in the `db.json` file.

Use Fetch to Get and Display Tasks from the API

Now that we have the database and the interface ready, the next step is to use JavaScript to get tasks data and display it in our table. We're going to use the Fetch API for that.

Open the `script.js` file, and write the following `getTask()` function:

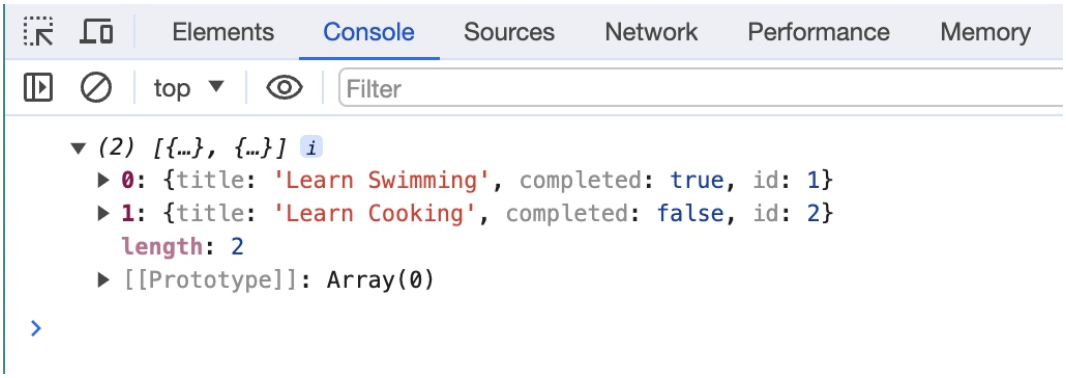
```
const BASE_API_URL = 'http://localhost:3000/tasks';

function getTasks() {
```

```
  const tableElement = document.querySelector('#tasks-table');

  fetch(BASE_API_URL)
    .then(response => response.json())
    .then(tasks => {
      console.log(tasks);
    });
}

getTasks();
```

In the code above, we declared a constant variable to store the API URL we're going to hit with requests.

Below the constant, we created a function called `getTasks()` that will execute a network request using `fetch()`. To check if we successfully retrieved the data, we printed the `tasks` to the console.

If you open the browser console, you should see the following output:



This means we successfully retrieved the tasks data. The next step is to display those data on our To-do List. We need to query the `<tbody>` element in our HTML page and put the tasks as rows there.

Recall that in our previous chapter, we put the `id` attribute to the `<tbody>` tag as follows:

```
<tbody id="tbody-tasks">
  ...
</tbody>
```

This means we can get the element using the query selector:

```
const BASE_API_URL = 'http://localhost:3000/tasks';

const tableElement = document.querySelector('#tbody-tasks');
```

Then we go back into the `getTasks()` function, remove the `console.log()` method and create table rows based on the `tasks` data.

Below, you can see that the code for creating the elements is quite long, but it's essentially the same as when we create buttons for the Wizard Form:

```
function getTasks() {
  let tableRows = '';

  fetch(BASE_API_URL)
    .then(response => response.json())
    .then(tasks => {
      tasks.forEach(task => {
        const element = `<tr class=${
          task.completed ? 'table-success' : ''
        }><td><input class="form-check-input" type="checkbox" ${
          task.completed ? 'checked' : ''
        } onclick="toggleTask(event, ${task.id})"></td><td>${
          task.title
        }</td><td><button type="button" class="btn btn-link"
onclick="editTask(${
          task.id
        })">Edit</button></td><td><button type="button" class="btn btn-link"
onclick="deleteTask(${
          task.id
```

```
        })">Delete</button></td></tr>`;

        tableRows += element;
      });
      tableElement.innerHTML = tableRows;
    });
}
```

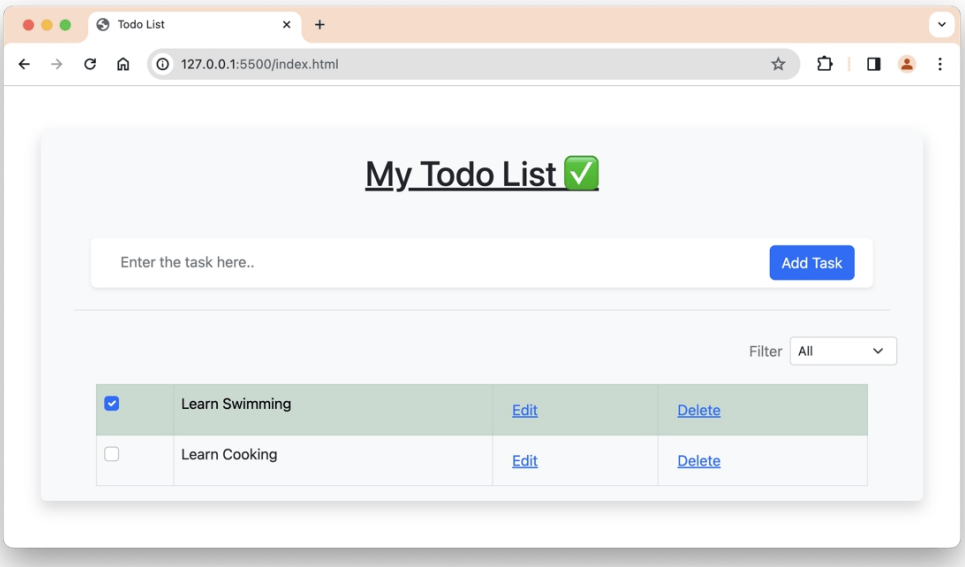
In the above code, we create a variable named `tableRows` that starts as an empty string. Next, we iterate over the `tasks` array using the `forEach()` method, and create an `element` variable for each `task` data.

This `element` variable uses the data from the array to dynamically adjust the HTML output. If the task is marked as completed, then the `table-success` class is added to the `<tr>` tag, and the `checked` attribute is added to the checkbox.

When the checkbox is clicked, then the `toogleTask()` function will be called. When the Edit button is clicked, then the `editTask()` function will be called, and when the Delete button is clicked, the `deleteTask()` button is called.

The elements are stored in the `tableRows` variable, and when the `forEach()` process is finished, we set the `innerHTML` value of the `tableElement` to the `tableRows` value.

If you open the browser, you should see the tasks appear on the table like this:



Since we're able to display the data, we can remove the hard-coded `<tr>` elements in our `index.html` file. Remove the highlighted lines below:

```
<!-- remove this from index.html -->
<tr>
  ... Learn JavaScript ...
</tr>
<tr>
  ... Learn HTML ...
</tr>
```

Now that we successfully displayed the data, the next step is to enable adding a new task to the database.

Adding a New Task

To add a new task to the database, we need to define the `addTask()` function that has been defined in the previous chapter for the 'Add Task' button:

```
<button type="button" class="btn btn-primary" onClick="addTask()">
  Add Task
</button>
```

Back to the `script.js` file, create a new function named `addTask()` and write the following code into it:

```
function addTask() {
  const taskTitle = document.querySelector('#task-title');

  const data = {
    title: taskTitle.value,
    completed: false,
  };

  fetch(BASE_API_URL, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  }).then(response => {
    if (response.ok) {
      alert('Task added');
      getTasks();
      taskTitle.value = '';
    }
  });
}
```

First, we query the input text, which has an id of `task-title`, then we create an object called `data` that stores the `title` and `completed` status of the task.

After that, we send a POST request using `fetch()`, adding the required options. When the response comes, we call the `alert()` function to notify the user that the task has been added to the database.

Because there's a new task on the database, we called the `getTasks()` data to reload the task lists and display it to the user. Then we empty the input text just to make it intuitive.

Now you can test adding a new task from the browser. The next step is to add the delete functionality.

Deleting a Task

To delete a task from the database, we need to define the `deleteTask()` function in our `script.js` file. Here's the function in full:

```
function deleteTask(id) {
  const confirmation = confirm('Are you sure you want to delete the task?');
  if (confirmation) {
    fetch(BASE_API_URL + '/' + id, {
      method: 'DELETE',
    }).then(response => {
      if (response.ok) {
        alert('Task deleted');
        getTasks();
      }
    });
  }
}
```

First, we confirm the decision to delete the task with the user using the `confirm()` function, which is part of the browser API.

When the user confirms, we send a DELETE request using the `fetch()` function, and we put the `id` value of the task we want to delete in the API URL (`BASE_API_URL + '/' + id`)

When we got a response back, we just call the `alert()` function to notify the user, then refresh the tasks list with `getTasks()`.

Now You can test the delete button from the browser. It should be working.

Editing a Task

To edit a task, we need to declare the `editTask()` function in our `script.js` file as follows:

```
function editTask(id) {
  const newTitle = prompt('Enter the new task title');
  if (newTitle) {
    const data = { title: newTitle };
    fetch(BASE_API_URL + '/' + id, {
      method: 'PATCH',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(data),
    }).then(response => {
      if (response.ok) {
        alert('Task updated');
        getTasks();
      }
    });
  }
}
```

To edit a task, we need to call the `prompt()` function to ask the user a new title for the task. When the new title is received, we put the title as an object inside the `data` variable.

The next step is to perform a network request using `fetch()`, add the `id` of the task to the URL, and set the method to PATCH.

When we receive a response, we notify the user and refresh the list.

Marking a Task as Completed

To mark a task as completed or not completed, we need to check if the checkbox input has the `checked` attribute.

To do so, we need to access the `event` property when the input is clicked. This is why we pass the `event` property on the checkbox `onclick` attribute:

```
<input ... type="checkbox" onclick="toggleTask(event, id)">
```

Create a `toggleTask()` function in the `script.js` file as shown below:

```
function toggleTask(event, id) {
  const checked = event.target.checked;
  const data = {
    completed: checked,
  };
  fetch(BASE_API_URL + '/' + id, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  }).then(response => {
    if (response.ok && checked) {
      alert('Hurray! You finished the task');
    }
    getTasks();
  });
}
```

The `checked` attribute returns `true` when the checkbox is checked, or `false` otherwise.

We wrap this value in a `data` object and then send a PATCH request using `fetch()`, similar to the `editTask()` function we

created earlier.

When the task is marked as completed, we notify the user with an `alert()` call. To refresh the list, we call the `getTasks()` function.

Now we have all core functionalities created using JavaScript. The last step is to enable the filter function.

Filtering Tasks

The select input that we've created will serve as a filter function. We can refresh the list to see only completed or active tasks.

In JSON server, you can filter the returned data by adding a query parameter. To filter by the `completed` property value, the URL is http://localhost:3000/tasks?completed=true or http://localhost:3000/tasks?completed=false.

You can test this by opening the URL on the browser.

Since we already have a `getTasks()` function, we can create a `filterTasks()` function that simply calls on `getTasks()`:

```
function filterTasks(event) {
  const filterValue = event.target.value;
  if (filterValue === 'all') {
    getTasks();
  } else if (filterValue === 'completed') {
    getTasks('completed');
  } else {
    getTasks('active');
  }
}
```

Here, we use the `event` property again to get the value of the select input.

Next, we check the value of the `filterValue` and add the relevant argument when calling the `getTasks()` function.

If we want to see only completed tasks, then we send the 'completed' value to the `getTasks()` function.

To accommodate the filter argument, we need to add a parameter to the `getTasks()` function. Let's update the function as shown below:

```
function getTasks(filter) {

  let parameter = '';
  if (filter === 'completed') {
    parameter = '?completed=true';
  } else if (filter === 'active') {
    parameter = '?completed=false';
  }

  const API_URL = BASE_API_URL + parameter;
  fetch(API_URL)
  ...
}
```

Here, we add the `filter` parameter to the `getTasks()` function. Then, we create a query parameter for the URL using an `if-else` check.

Next, we add the `parameter` to the `BASE_API_URL` and use it when calling the `fetch()` function.

This way, the GET request corresponds to the filter value you selected.

And now the application is complete! You can try all the functionalities on your browser.

If you encounter an error and need some help, you can compare your code with mine at https://github.com/nathansebhastian/js-todo-list/tree/main/check-point-2 or email me at nathan@codewithnathan.com

Summary

Well done! You've just finished your second JavaScript project. Hopefully, this project has shown you how JavaScript is the secret ingredient that makes a web application interactive.

In JavaScript, we define functions that respond to user actions, and then plug those functions into the HTML `onclick` attribute.

When the user performs an action, JavaScript will respond by running the relevant process. The CRUD operations are the core functionalities of a web application, and you'll see them no matter what applications you use.

WRAPPING UP

Congratulations on finishing this book! It wasn't easy, but you did it anyway.

I hope this book has improved your understanding of JavaScript and the role it plays in software development, particularly when it comes to front-end development.

Now you see how JavaScript can be used to manipulate what's being shown to the browser window long after the HTML and CSS part of the site has been rendered.

The Browser API consists of a set of objects and methods that you can access using JavaScript, allowing you to manipulate the content of your website. The `Document` object allows you to select specific elements using the `querySelector()` method.

The browser also exposes signals known as events that you can listen to. This way, you can execute some code as a reaction to the event, such as the 'click' and 'submit' events.

The Next Step

There are other APIs that you can use to get the location of the user, accept images and files submitted by the user, or access your camera and microphones.

Exploring all these APIs is beyond the scope of this book, so I'll make another book where we're going to explore the Browser APIs in full and build projects with it.

You can also tackle learning popular JavaScript libraries such as React, Vue or Svelte next as you now have the fundamental knowledge of how JavaScript works. I'm also planning to write a book on these libraries.

If you enjoy this book, you might consider subscribing to my newsletter to know when I release a new one at https://codewithnathan.com/newsletter

I also have a 7-day free email course on Mindset of The Successful Software Developer which is free at https://sebhastian.gumroad.com/l/mindset

Each email unveils a slice of wisdom that I got after working as a web developer and programmer for 8+ years in the tech industry, from startups to mega-corporations. It offers a fresh perspective on what it means to be a successful software developer.

If you like the book, I would appreciate it if you could leave me a review in Amazon too because it would help others to find this book.

For any questions, feel free to email me at nathan@codewithnathan.com.

If you didn’t like the book, or if you feel that I should have covered certain additional topics, please let me know in the email. This book can only get better thanks to readers like you.

Thank you and all the best for your career in tech!

Until next time.

ABOUT THE AUTHOR

Nathan Sebhastian is a senior software developer with 8+ years of experience in developing web and mobile applications.

He is passionate about making technology education accessible for everyone and has taught online since 2018

Beginning Modern JavaScript

A Step-By-Step Gentle Guide to Learn JavaScript for Beginners

By Nathan Sebhastian

https://codewithnathan.com

Copyright © 2023 By Nathan Sebhastian

ALL RIGHTS RESERVED.

No part of this book may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission from the author.