SIMPLIFIED JAVASCRIPT FOR VIPS

No fuss,
No time wasted!
Learn JavaScript Quickly

EBENEZER DON

Simplified JavaScript for Very Important Programmers

The Fast Track to Mastering Essential JavaScript Concepts

Ebenezer Don

This book is for sale at http://leanpub.com/vip-javascript

This version was published on 2023-03-10

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Getting The Book

I'm thrilled that you're interested in this book! To get your hands on a legitimate copy, I highly recommend visiting vipjavascript.com*, Leanpub.com/vip-javascript† or selar.co/vip-javascript‡.

By purchasing from the official websites, you'll receive access to the latest updates and corrections, and you'll be supporting my hard work as an author.

I kindly ask that you refrain from sharing or pirating the book. I've put in countless hours of work, and it's important that I'm compensated for my efforts.

Thank you for your support, and happy reading!

*https://vipjavascript.com
†https://leanpub.com/vip-javascript
‡https://selar.co/vip-javascript

Introduction

"Simplified JavaScript for Very Important Programmers" is a refreshing and practical guide for beginners who want to master every essential aspect of JavaScript. In this book, I offer a streamlined approach to learning JavaScript with relatable real-life examples that make the concepts easy to understand, even for those without programming experience.

I've divided this book into two parts. The first part is designed for everyone learning JavaScript, and the second part focuses on using JavaScript on the web, building on the foundational concepts covered in the first part.

I've written everything in clear and concise language, making it easy for everyone to follow along and grasp. Whether you're a student, an entrepreneur, or simply someone interested in programming, this book will guide you through the essential concepts of JavaScript with ease and efficiency.

From variables and operators to loops, functions and objects, we'll cover everything you need to know to become a competent JavaScript programmer. The examples and exercises throughout this book are designed to reinforce your learning and help you apply your knowledge in real-world scenarios.

If you're looking for a practical and enjoyable way to learn JavaScript, you're reading the perfect book.

What is JavaScript and How Does it Work?

JavaScript is a programming language that was initially developed as a means to add dynamic and interactive elements to websites. Today, it is one of the most popular programming languages in the world and is used for a wide range of applications, including web development, mobile app development, server-side scripting, machine learning, game development, and more. When learning web development, it is essential to understand how JavaScript works and how you can use it to build dynamic websites and add interactivity to your web pages.

JavaScript is a dynamically-typed language, which means that you can use it to store different types of information without telling the computer what you'll put there ahead of time. Since you don't have to keep track of so many rules, it's usually easier to read and write JavaScript than other programming languages like C++, Java, or Rust. However, this also means that you can make mistakes that are hard to catch, so it's important to be careful when writing JavaScript code.

JavaScript is often used in conjunction with HTML and CSS to create interactive and dynamic websites. One of its main features is its ability to manipulate and interact with web pages; it can be used to change the content and layout of a webpage, respond to user input, and communicate with servers to retrieve and update data. So, while HTML provides the structure and content of a webpage, and CSS defines its style and layout, JavaScript adds behavior and interactivity.

How Does JavaScript Work?

When a JavaScript program is run, the computer reads the code and follows its instructions.

In the case of web development, JavaScript is usually run in a web browser like Google Chrome or Mozilla Firefox. When you open a webpage that contains JavaScript, your browser reads the JavaScript code and uses it to interact with and modify the content of the webpage, its layout, and its behavior.

You can also use JavaScript in other contexts like mobile app development and server-side programming, in which case, the JavaScript code is run on a different type of device or platform, but the process of executing the code is similar.

One thing to note is that JavaScript is an interpreted language, meaning it is not compiled (translated into machine code) before it is run. Instead, the computer interprets the code as it is being executed. This makes writing and debugging JavaScript programs easier, but it can also make them run slower than programs written in compiled languages.

History of JavaScript

JavaScript was created by Brendan Eich in 1995 and was first introduced with the release of Netscape Navigator 2.0, a popular web browser at the time. JavaScript was initially called Mocha but later renamed to LiveScript and then to JavaScript in an attempt to take advantage of the buzz surrounding the Java programming language.

JavaScript quickly gained popularity among web developers and became supported by other major web browsers like Internet Explorer and Safari. In 1996, JavaScript was standardized by the European

Computer Manufacturers Association (ECMA) as ECMAScript, which remains its official name today.

Differences Between JavaScript and HTML/CSS

JavaScript is often used with HTML and CSS to build websites. However, there are some important differences between the three languages. One crucial difference is that JavaScript is a programming language, while HTML is a markup language and CSS is a style sheet used to describe the appearance of a webpage. So, you'll often find JavaScript more difficult to learn than HTML and CSS. Some key features of JavaSCript that are not found in HTML and CSS include:

- The ability to execute code based on conditions or events
- The ability to iterate through data and perform actions on multiple elements
- The ability to create and manipulate different types of data
- The ability to make requests to servers and retrieve data

While HTML and CSS are important for defining the content and appearance of a webpage, without JavaScript, it's difficult to add interactivity to your webpage that'll make users love it. JavaScript allows developers to build more sophisticated and engaging user experiences, and it is a critical component of modern web development.

Uses of JavaScript across multiple fields

Apart from web development, you can use JavaScript for a wide range of applications, including:

Server-side programming

When you visit a website, your web browser sends a request to the website's server, which then executes the code responsible for handling that request and generating an appropriate response. Tools like Node.js and Deno make it possible to run JavaScript on the server instead of a browser. This enables developers to build scalable, high-performance applications that can handle many concurrent users.

Mobile app development

You can use JavaScript to build cross-platform mobile apps with tools like Cordova, React Native, and NativeScript. These tools allow developers to write JavaScript code that is compiled and run on a mobile device rather than in a browser.

Game development

You can also use JavaScript to build browser-based games with tools like Phaser and Babylon.js. These tools allow you to create 2D and 3D games that can be played in a web browser.

Data visualization

JavaScript libraries like D3.js and Highcharts can be used to create interactive data visualizations for websites and applications.

Robotics

You can use JavaScript to control and program robots using platforms like NodeBots and Johnny-Five.

Internet of Things (IoT)

JavaScript can be used to build applications that interact with and control connected devices, such as smart home appliances and sensors.

Machine learning

With JavaScript libraries like TensorFlow.js and Brain.js, you can build machine-learning models and incorporate them into web applications.

How to write and run JavaScript code

The best way to learn JavaScript is by writing JavaScript. Let's explore the different ways you can write and run JavaScript code so that it'll be easy for you to practice whatever you learn.

Using a web browser

One of the simplest and most accessible ways to write JavaScript code is by using a web browser. Most modern web browsers, such as Google Chrome, Mozilla Firefox, and Safari, have a built-in JavaScript engine that allows you to write and run JavaScript code directly in the browser.

To write JavaScript code in a web browser, you can use the browser's developer tools. To open the developer tools in Google Chrome, right-click on any webpage and select "Inspect" from the context menu. Alternatively, you can press `Ctrl + Shift + I` on a Windows or Linux computer, or `Command + Option + I` on a Mac.

Once the developer tools window is open, navigate to the "Console" tab, where you'll be able to write your JavaScript code. For example, you can try typing the following code into the console, but with your name instead of mine:

```
1  console.log('My name is Ebenezer Don')
2  console.log('JavaScript is awesome!')
```

To run your JavaScript code, press the Enter key on your keyboard. This will print the message "My name is Ebenezer Don" and

"JavaScript is awesome!" to the console. Good job! I'll explain every part of the code you just wrote later on.

Using an online code editor

Another way to write and run JavaScript code is by using an online code editor or online Integrated Development Environment (IDE). There are many online code editors and IDEs available that allow you to write and run JavaScript code from your web browser. Some popular options include CodePen, JSFiddle, and JS Bin.

You need an internet connection to use an online code editor, but it has several advantages too. First, it allows you to write and run code from anywhere. Second, you can easily share your code with others through links. Third, many online code editors and IDEs have built-in features that can ease the process of writing code and checking for errors.

Using an offline code editor

There are also several offline code editors and IDEs you can use to write and run JavaScript code. Some popular options include Eclipse, NetBeans, Visual Studio and Visual Studio Code (with the help of extensions). These IDEs need to be installed on your local computer and typically have more features and functionality than online code editors. They are often used for more complex projects, and even if you don't start with them, you'll eventually need one as you progress.

The easiest to get started with is your browser console, so I recommend writing and running your first JavaScript code on it, and when you do, take a screenshot and save it where you can access it a year from now. I'm sure you'll be proud of your progress.

JavaScript syntax and Data types

JavaScript syntax and data types are fundamental concepts that you should understand as you learn JavaScript. The syntax of a programming language includes its rules for writing and structuring code, while data types represent the classification of values that you can store and manipulate in your code.

Syntax

Here are some basic rules for writing and structuring JavaScript code:

- JavaScript is case-sensitive, which means that the case of the characters and keywords you use matters. So if you write a word like 'name' in your code, JavaScript will see it as different from 'Name' or 'nAme'. This is a common confusion for beginners when writing code, so pay close attention to the cases you use.
- Use spaces, tabs, and newlines to make your code readable, but make sure to stay consistent. This is also known as **indentation** and you'll see examples as we progress.
- You can choose to write a semicolon (;) at the end of each statement. *A statement is any line of code that instructs the computer to perform a specific task.* Terminating statements with semicolons used to be a requirement for JavaScript code to run properly but that isn't the case anymore. So you'll see some codebases with semicolons and some without it.

> It's important to keep your code style consistent as this will ease readability and understanding, so if you choose to use semicolons, use them for every statement in your code, and if you choose to do without semicolons, don't use them at all.

There are some more rules that won't be easy for you to understand now, so I'll show them to you as we progress.

Data Types

In JavaScript, there are several data types that are used to represent different types of information, like numbers, text, or true/false values. The most basic data types are numbers, strings, and Booleans. A collection of multiple values with the same or different data types is called a data structure.

JavaScript is a dynamically-typed language, which means that the type of a value is determined while the program is running and can change during the program. This is different from other programming languages where you have to set the type of a value when creating, and can't change it later. These other languages are called "statically-typed".

There are several basic data types in JavaScript, including:

Numbers

Numbers in JavaScript can be integers or floating-point values. Integers are whole numbers like 1, 2, or 3, and floating-point values are numbers with decimal points like 7.8 or 0.05.

You've already used `console.log()` to display your name in the browser console. `console.log()` is also a **method**, which we'll explain in detail later on. For now, you can recognize methods with

the parenthesis() after them. They represent a set of instructions for the computer.

Let's introduce another method, `typeof()`. We'll use this to find out the type of any value we create in JavaScript. On your browser console, writing the following code and hitting the Enter key should display the text "number":

```
typeof(10)
```

It's important that you practice everything you learn to make understanding and remembering it easier. Try using the`tyepof()` method to find out the type of **-10**.

Notice that you still get the response, 'number'. This is because there is no difference between positive and negative numbers. Positive numbers are numbers greater than zero, and negative numbers are less than zero. You can represent both positive and negative numbers with the same syntax in JavaScript.

Strings

In JavaScript, a string is a group of characters that is used to store text. Strings are commonly used to store words and sentences. To write a string, you have to wrap it in either single quotes ('This is a string') or double quotes ("This is also a string"). Both types of quotes can be used to write strings, as long as they are used consistently. So, if you start writing a string with single quotes, you must end it with single quotes.

A string can be a single word, sentence or multiple paragraphs, and It can contain any combination of letters, numbers and spaces.

The text "My name is Ebenezer Don" in the `console.log()` statement we used earlier is also a string. Remember to always use your name wherever you see "Ebenezer Don".

Let's print the string "Hello World" to the console, using both single quotes and double quotes:

```
console.log('This is really cool!')
console.log('This is really cool!')
```

Now, try using the `tyepof()` method to find out the type of "Hello World". Remember to pay close attention to the casing of keywords. For example, there's no uppercase letter in the `typeof()` method.

The string is an important data type in JavaScript. You can use it to represent text-based data like names, messages or addresses.

Booleans

Booleans represent true or false values. They are often used to make comparisons and to determine whether a set of instructions should be executed. When you run the code `tyepof(true)` in your browser console, you should get the response, 'boolean'.

To make value comparisons in JavaScript, you can use the symbols; > (greater than), < (less than), or === (is the same as).

Now, let's see if the strings "Hey" and 'hey' are the same in JavaScript. Run the following line of code in your browser console:

```
'hey' === 'HEY'
```

Running the above code should return **false**, and that's because strings are case-sensitive in JavaScript.

Booleans are an important data type in JavaScript, as they allow you to represent logical values, make comparisons and run code conditionally.

Comments in JavaScript

In JavaScript, comments are pieces of text within the code that are ignored when the code is executed. They are useful for adding

notes to your code, or for temporarily disabling a piece of code without having to delete it. There are two ways to write comments in JavaScript: Single-line comments and Multi-line comments.

Single-line comments

Single-line comments start with two forward slashes (`//`) and continue until the end of the line. For example:

```
// This is a single-line comment
```

Multi-line comments

Multi-line comments start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). They can span multiple lines and can be used to disable blocks of code or to add longer notes and explanations to your code. Here's an example of a multi-line comment:

```
/*
  This is a
  multi-line comment
*/
```

It's a good practice to make your code readable enough that you don't need to add comments to explain what it does. However, comments are useful when you want to explain why you did something a certain way, or when you believe that your code might be confusing to someone else or even your future self. Overuse of comments can make your code cluttered and difficult to read, so it's important to strike a balance between writing readable code and adding comments where necessary.

Basic conditional statements (If-else)

In JavaScript, you can use conditional statements to run different blocks of code depending on whether or not a condition is met. *A block of code is a group of statements enclosed in curly braces.*

Conditional statements are a crucial aspect of programming, as they allow you to control the flow of your program and make it behave differently in various situations.

There are two main types of conditional statements in JavaScript: `if` statements and `switch` statements. In this section, you'll learn about `if` statements, and later on, `switch` statements.

The If Statement

An if statement enables you to run a block of code if a specific condition is true. This is the basic syntax for an if statement:

```
1  if (condition) {
2    // code to run if condition is true
3  }
```

The condition is a boolean expression that evaluates to either true or false. An expression is a combination of values and operators that evaluates to a single value. For example, `10 > 5` is an expression that evaluates to true, and `10 < 5` is an expression that evaluates to false. An expression can also be a single value, like a number, string, or boolean.

If the condition evaluates to true, the code block inside the **`if` statement** is executed. If the condition evaluates to false, the code block is skipped. The code block is enclosed in curly braces `{}`and is called the **`if` body**.

For instance, running the following code should return the string "7 is greater than 5":

```
if (7 > 5) {
  console.log('7 is greater than 5')
}
```

SInce 7 is greater than 5, the `console.log()` statement in the curly braces will run. Likewise, running the following code will not return anything on your console:

```
if (7 < 5) {
  console.log('7 is less than 5')
}
```

Notice that we changed the comparison symbol from > (greater than) to < (less than). As 7 is not less than 5, the computer will not execute the console.log() statement.

You'll also notice that the code block inside the curly braces has two spaces before it. This is called **indentation**, and it makes code easier to read and understand. We indent code to show that it is inside a block. If we didn't indent the code inside the `if` statement, it would be hard to tell whether it's part of the `if` statement or whether it's a separate statement entirely Indentation is not required for JavaScript to run, but it's necessary for writing readable code and avoiding errors. You can use any number of spaces for indentation, but it's common to use two spaces. You'll find some developers who use four spaces or tabs, but it's best to stay consistent with whichever style you or your team choose.

The Else clause

You can add an **else clause** to an if statement, which allows you to specify a block of code to run if the condition is false. This is the basic syntax for an if-else statement:

```
if (condition) {
  // code to run if condition is true
} else {
  // code to run if condition is false
}
```

Let's add an else clause to our conditional statement:

```
if (7 < 5) {
  console.log('7 is less than 5')
} else {
  console.log('The condition is false')
}
```

Since 7 is not less than 5, running the above code should return the string "The condition is false".

Conditional statements are an essential tool for creating programs that can adapt to different circumstances and make decisions based on the data they receive. We'll cover more complex conditional statements and other ways we can write them as we progress.

Variables in JavaScript

Introduction

It's not easy to write JavaScript code without using variables. So far, we've done an excellent job of avoiding them, but it's time you learned about one of the most important concepts in programming.

A **variable** is a named location for storing data in the computer's memory. In simple words, it's a container in which we can store all kinds of data, like numbers, text, pieces of code, and even other variables. So, think of a variable as a box with a label on it. The label is the variable's name, and the content of the box is the variable's value.

Here's an example of a variable in JavaScript:

```
1  let myName = 'Ebenezer Don'
```

In this example, we created a variable named "myName" and stored the string value "Ebenezer Don" inside it. This is called **variable declaration**. The variable declaration is made up of four parts: the declaration keyword, the variable name, the assignment symbol (=), and the variable value.

The declaration keyword

The keyword `let` is one of the ways you can declare a variable in JavaScript. There are other ways to declare variables, but `let` is the most common one. You'll learn about the other variable declaration keywords as we progress. *In JavaScript, a keyword is a reserved word with a specific purpose.*

The variable name

The variable name in this example is `myName`. It's a good practice to use descriptive names for your variables because when the name of your variable represents its value, your code is easier to understand.

The assignment symbol (=)

The assignment symbol (=) is used to assign a value to a variable. It's also called the assignment operator, and there are other operators in JavaScript, like the addition operator (+), the subtraction operator (-), the multiplication operator (*), and the division operator (/). You'll learn about them in the next chapter.

The variable value

In our example, the variable value is the string "Ebenezer Don." The value of a variable can be of any type, including numbers, strings, booleans, or other variables.

We can use the console.log() function to print the variable's value to the console. In addition to the code above, run the following console.log statement in your browser's console:

```
1  console.log(myName)
```

This will print the value of the variable `myName` to the console.

We can also reassign a variable's value by using the assignment operator (=). So if we want to change the value of the variable `myName` to "Jane Butters," we could do so by writing this line of code:

```
1  myName = 'Jane Butters'
```

Now, if you rerun the console.log statement, you'll get the string "Jane Butters" printed on the console.

Variable declaration and assignment

There are several ways to declare a variable in JavaScript, each with its use case.

The let keyword

The `let` keyword is the most common way to declare a variable in JavaScript. It's used to create local variables, which are variables that are accessible only within a particular block of code.

Here's an example of a local variable with the `let` keyword:

```
if (true) {
  let message = 'JavaScript is awesome!'
}

console.log(message)
```

In this example, the message variable is **local** and is only accessible within the block of code enclosed in the curly braces. So you'll get an error if you try to access it outside this block like we're doing in the `console.log` statement.

Global variables, on the other hand, are variables that are accessible from anywhere in your code. If you want to create a global variable with the `let` keyword, you can do so by declaring it outside your code blocks.

Here's an example of a global variable with the `let` keyword:

```
let name = 'Jane'

if (true) {
  console.log(name)
}
```

Running this code will return the string value "Jane" to the console. This is because the name variable is global and is accessible from anywhere in your code.

The var keyword

The `var` keyword is the oldest way to declare a variable in JavaScript and is used to create global variables. So, if we used the `var` keyword in our first example to declare the message variable, we wouldn't get an error.

Here's an example of a global variable with the `var` keyword:

```
if (true) {
  var message = 'JavaScript is awesome!'
}

console.log(message)
```

The above code will not return any errors because the message variable is global. This means that it is accessible from anywhere in the code, including outside the block of code in which it was declared.

It's important to be careful when using global variables. Since they can be accessed from anywhere in your code, global variables can cause problems. For example, if you declare a global variable with the same name as a local variable in your code, the global variable will overwrite the local variable; this can lead to unexpected results and confusion in your code.

The const keyword

So far, we've seen how to create local and global variables which can be reassigned. But most times, it's safer to create variables that can't be reassigned so that we don't accidentally change their values. The `const` keyword is used for this purpose.

Here's an example of a constant variable:

```
const age = 21
```

If we attempt to reassign the value of the age variable, we'll get an error:

```
age = 22

// Uncaught TypeError: Assignment to constant variable.
```

This is because the age variable is a constant variable and can't be reassigned. Using the `const` keyword is my preferred way of declaring variables that don't need to be reassigned.

Variable naming rules and conventions

In programming, a **convention** is a set of guidelines you should follow to ensure that your code is written in a consistent style and is easy to read and understand. When naming your variables, it's essential to follow the JavaScript variable naming conventions to ensure that your code runs properly and to avoid bugs. A **bug** is an error in your code that causes it to run incorrectly or not run at all.

Here are the rules and conventions commonly followed when naming variables in JavaScript. We'll start with the things you

should avoid, then move on to what you should do when naming your variables.

Avoid using numbers as the first character of a variable name

Variable names can't start with a number, although they can contain numbers. For example, the variable name `person1` is valid, but if you name your variable `1person`, you'll get an error.

Avoid using special characters in variable names

In JavaScript, special characters are characters that have a specific meaning or use in the language and are not letters, numbers, or symbols. These include the following: @ ! # $ % ^ & * ( ) _ + - = { } [ ] : ; " ' < > , . ? / \ | ` ~

Variable names cannot contain special characters, so if you try to name your variable `name-of-person` or `#name_of_person`, you'll get an error. The dollar symbol is sometimes used as a prefix for variable names, but this is not a requirement and is generally not recommended.

Avoid using spaces in variable names

Variable names cannot contain spaces. So naming your variable `my country` or `my age` will result in an error.

Avoid using reserved words in your variable names

JavaScript has a list of reserved words that cannot be used as variable names. These include words like `var`, `const`, and `let`. Using a reserved word as a variable name will result in an error. You can check out the full list of JavaScript reserved words in the Mozilla Developer Network (MDN) documentation. (link here)

Maximum and Minimum length of a variable name

There is no explicit maximum or minimum length for variable names in JavaScript. However, some constraints can affect the maximum length of your variable name.

One such constraint is the maximum length of a string in JavaScript, which is `2^53 - 1` (approximately `9` quadrillion characters). So you can't have a variable name longer than this, but it's unlikely that you'll need to use a variable name this long in practice.

Use camelCase for variable names

When naming variables, one word might not be enough to describe what the variable is storing. Since you cannot use spaces in variable names, you can use camelCase to separate the words that make up your variable name.

CamelCase is a naming convention in which the first letter of a variable name is in lowercase, and the first letter of each subsequent word is in uppercase. For example, if you want to name a variable that stores a student's age, you can write it as `studentAge` or `ageOfStudent`.

In JavaScript, camelCase is the most commonly used naming convention for variables. So, when naming your variables, always start with a lowercase letter and use camelCase to separate the words.

Use meaningful variable names

It's important to use descriptive and meaningful names for your variables; this will make your code easier to write, read and understand. For example, if you're creating a variable to store a student's name, you should write something like `nameOfStudent` or `stuendentName` instead of `x` or `sname`. Your variable names should always represent the data they store because descriptive variable names make it easier to understand the purpose and function of variables. This will help you avoid bugs, improve the quality of your code and make it easier to maintain.

How you name your variables is very important, and by following the recommended conventions, you'll be able to write code that is easy to read, understand and maintain.

Basic JavaScript operators

Operators are symbols that perform actions on values. For example, when you write `2+3` on your calculator, you use the `+` operator to add two values. The values `2` and `3` are called operands, while the plus symbol (`+`) is the operator. In this chapter, we'll explore the different types of operators in JavaScript, including:

- Arithmetic operators
- String operators
- Assignment operators
- Comparison operators
- Logical operators

Arithmetic operators

We use arithmetic operators to perform mathematical calculations like addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`).

Here are examples you can try in the console:

```
console.log(10 + 2) // Output: 12
console.log(10 - 2) // Output: 8
console.log(10 * 2) // Output: 20
console.log(10 / 2) // Output: 5
```

We also have the increment and decrement operators. The increment operator (`++`) increases the value of a variable by `1`, while the

decrement operator (—) decreases the value of a variable by 1. These operators can be placed before or after the operand.

When placed before the operand, the operators are called prefix operators, which means that the operation is applied before the value is returned, giving us a new value. Here's an example:

```
let firstValue = 3
console.log(++value) // Output: 4
```

When placed after the operand, they are called postfix operators; the operation is applied to the variable after the current value is returned. Here's an example:

```
let secondValue = 5
console.log(value++) // Output: 5
console.log(value) // Output: 6
```

Notice that we have to log the variable separately to get its new value. This is because when you use the postfix operator, the value is returned first, and then the operation is applied.

It's also important to note that the increment and decrement operators reassign new values to variables, so you can only do this with variables you declare using the `let` keyword. If you try to use these operators on a variable you declare with the `const` keyword, you'll get an error. Let's give it a try. Remember always to run these examples yourself to see the results:

```
const thirdValue = 7
console.log(++value)
// Uncaught ReferenceError: value is not defined
```

There are two other arithmetic operators that you can use in JavaScript: **the remainder operator** (`%`) and **the exponentiation operator** (`**`).

The **remainder operator** returns the remaining value when an operand is divided by another. So if you run `5 % 2` on your console, you'll get **1** because 2 will go into 5 twice and leave 1 as the remainder.

The **exponentiation operator** raises the first operand to the power of the second operand. So if you run `2 ** 3` on your console, you'll get **8** because 2 to the power of 3 is 8 (that is 222).

The Concatenation operator

In JavaScript, we use the concatenation operator to concatenate strings. The plus symbol (+) serves as the concatenation operator when one or more of the operands are strings. So if you run `'Hello' + 'World'` on your console, you'll get "HelloWorld" as the output. Also, if you run `'Hello' + 1` on your console, you'll get 'Hello1' as the output. When you use the plus operator with strings and numbers, the numbers will be converted to their string form and then concatenated. Let's try some more examples:

```
1  console.log('Hello' + ' ' + 'Everyone')
2  // Output: Hello Everyone
3  console.log('James ' + 'Bond ' + 007)
4  // Output: James Bond 007
5  console.log(
6    10 + 'years from now' + "I'll be a 100 years old"
7  ) // Output: 10 years from now I'll be a 100 years old
```

Notice that we used the space character (' ') to separate the words in the first example. If we didn't use the space character, we would get 'HelloEveryone' as the output.

In the second example, we added the space character in the same string as the word 'James'. This is another way to add spaces between words when concatenating strings.

In the third example, we used double quotes (" ") to enclose the string. This is because the string (I'll) contains a single quote which will cause an error if we also use single quotes (' ') to it.

Assignment operators

We use assignment operators to assign a value to a variable. You've already seen the assignment operator (=) in action. There are also compound assignment operators in JavaScript, which include: `+=`, `-=`, `*=`, `/=`, and `%=`. These compound operators perform an operation with a variable and an operand and then assign the result to the variable. For example, if you have a variable named "age" with a value of 10, and you want to increment it by 5, you can use the `+=` operator like this:

```
let age = 10
age += 5 // Output: 15
```

Using the `+=` operator is the same as writing `age = age + 5`, which reassigns the age variable to `10 + 5`. Let's try some more examples with the other compound assignment operators:

```
age = 10
age *= 5 // Output: 50
age /= 5 // Output: 10
age -= 5 // Output: 5
age %= 5 // Output: 0
```

Compound assignment operators are a great way to make your code less repetitive.

Comparison operators

We use comparison operators to compare two values and return a boolean value (true or false) based on the comparison result. Here are the comparison operators and their meanings:

- == : equal to
- != : not equal to
- > : greater than
- < : less than
- >= : greater than or equal to
- <= : less than or equal to
- === : strict equal to (compares both the value and the type of the operands)
- !== : strict not equal to (compares both the value and the type of the operands)

Let's try some examples:

```
console.log(10 == 10)
// Output: true
console.log(10 == 5)
// Output: false
console.log(10 != 5)
// Output: true
console.log(10 > 5)
// Output: true
console.log(10 < 5)
// Output: false
console.log(10 >= 10)
// Output: true
console.log(10 >= 11)
// false: 10 is neither greater than nor equal to 11
console.log(10 <= 10)
```

```
// true: 10 is equal to 10
console.log(10 <= 9)
// false: 10 is neither less than nor equal to 9
console.log(10 === 10)
// true: 10 is equal to 10 and they are both numbers
console.log(10 === '10')
// false: number 10 is not the same as string 10
console.log(10 !== 9)
// true: 10 is not equal to 9
console.log(10 !== '10')
// true: number 10 is not the same as string 10
```

Logical operators

In JavaScript, we use logical operators to perform logical operations like AND (`&&`), OR (`||`), and NOT (`!`). Here are some examples:

```
console.log(true && true) // Output: true
console.log(true && false) // Output: false
console.log(false && true) // Output: false
console.log(false && false) // Output: false

console.log(true || true) // Output: true
console.log(true || false) // Output: true
console.log(false || true) // Output: true
console.log(false || false) // Output: false

console.log(!true) // Output: false
console.log(!false) // Output: true
```

The AND operator returns true if both operands are true, and false if either operand is false. So if you run `console.log(10 > 5 && 7 < 10)` on your console, you'll get true as the output because 10 is greater than 5, and 7 is less than 10. But if you run `console.log(10`

`> 5 && 7 > 10)` on your console, you'll get false as the output because 10 is greater than 5, but 7 is not less than 10. When using the AND operator, both operands must be true for the result to be true.

The OR operator returns true if any of the operands is true, and false if both operands are false. So if you run `console.log(10 > 5 || 7 > 10)` on your console, you'll get true as the output because one of the operands is true (10 is greater than 5). If you run `console.log(10 < 5 || 7 > 10)` on your console, you'll get false as the output because both operands are false.

The NOT operator returns true if the operand is false, and false if the operand is true. So you can read `!true` as "not true", and `!false` as "not false".

Operators are an essential part of programming, and you'll always need them when writing code. Remember that you must practice a lot to become a good JavaScript developer. So go over this chapter a hundred times if you need to, and try using each of the operators with different examples. You'll get better at them with time.

Functions

In JavaScript, functions are a way to group code that can be reused or executed as many times as needed in a program.

You've already used the `console.log()` method to display things in your console, and the `typeof()` method to find the type of values. In JavaScript, built-in functions are called methods.

The console.log() statement contains several lines of code you don't see, and this is what actually tells the JavaScript engine to display the value you put in parenthesis on the console so that when you run `console.log('Hello World!')`, the JavaScript engine will display the text Hello World!

With JavaScript functions, you can run the same set of statements from multiple places in a program without having to rewrite it each time.

How to define functions in JavaScript

JavaScript allows us to define functions in different ways. A common way to do this is through the function keyword. Let's write a function that multiplies any number by two and returns the result:

```
1  function multiplyByTwo(value) {
2    const result = value * 2
3    return result
4  }
```

From our above function;

- The name of our function is `multiplyByTwo`, which is provided right after the `function` keyword.
- The parenthesis next to the **function name** holds the function parameters.
- **Parameters** are names used to represent the values we want to use in our functions.
- The code block after the opening curly brace is the **function body**.
- The `return` keyword is used to end the execution of a function while returning a specified value. This value is called the **return value**.

Function parameters

The parameters of a function are names used to represent real values we intend to provide when we call a function. In our example above, our function parameter is `value`. When we call the `multiplyByTwo` function, we'll provide the actual value for this parameter.

What is a function call?

A **function call** or **function invocation** is a request to execute a function. We'll see how to do this when we use the `multiplyByTwo` function.

The return keyword

We use the `return` keyword to end the execution of a function while providing a specified value. In our case, the `multiplyByTwo` function returns the calculation result. Let's see how to use the return keyword to end the execution of our function conditionally:

```
function multiplyByTwo(value) {
  if (isNaN(value)) {
    return ' Value must be a number'
  }

  const result = value * 2
  return result
}
```

In our new function, we added a condition that checks if the value parameter is a number, using the built-in JavaScript `isNaN` function. **isNaN** means "is not a number" and returns **true** if the provided value is not a number. In this case, if the `value` parameter is not a number, the first `return` statement would terminate the function and return the string, "**Value must be a number**".

The above syntax we used to define our JavaScript function is called **function declaration**. Now, let's tell our function to run the code it contains.

We'll do this by invoking it. And to invoke our JavaScript function, we only need to write its name followed by a parenthesis:

```
multiplyByTwo()
```

Remember that in our function definition, we used the parameter, `value` to represent the real value to be multiplied.

To provide that number to our function, we'll place it inside the `parenthesis` invoking our function:

```
console.log(multiplyByTwo(35))
```

Here's what we'll see on our console when we run the above code:

```
70
```

Let's try supplying our function with a non-number:

```
console.log(multiplyByTwo('this is a string'))
// output: Value must be a number
```

Function Parameters vs Arguments

When invoking our function, the values we supply to it in place of its parameters are known as `arguments`.

So in our case, the number **35** and the string "**this is a string**" are our function arguments.

- During a function definition, the names representing the values we intend to supply to the function are called `parameters`.
- During a function call, the actual we provide to the function are known as `arguments`.
- We can use up to `255` parameters in a function definition.

More on the return statement

Without using the `return` statement to tell our function what to return, its default value will be `undefined`.

Let's test this by creating a function without a `return` statement and then invoking it:

```
function multiplyByTwo(value) {
  const result = value * 2
}

console.log(multiplyByTwo(35))
// output: undefined
```

The function also returns `undefined` by default when we provide the `return` keyword without a value.This makes it an `**empty return**`:

```
function multiplyByTwo() {
  const result = value * 2
  return
}

console.log(multiplyByTwo(35))
// output: undefined
```

Anonymous functions

You've seen how to use function declarations to define named functions. You can also use function declarations to define **anonymous functions** and **Immediately Invoked Function Expressions (IIFE)**.

Anonymous functions are functions that are without names; hence, the word **anonymous**.

```
function () {
  return alert("I'm a function, and I'm anonymous!")
}
```

Since we cannot reference the above function by a name, there is no way to call it elsewhere in our codebase. When this is the

case, the function is usually an **IIFE** (Immediately Invoked Function Expression), which means that it'll be invoked immediately after it is declared. To achieve this, we'll wrap the function in parenthesis and immediately call it with another open-and-close parenthesis right after its declaration:

```
;(function () {
  return alert("I'm a function, and I'm anonymous!")
})()
```

Notice how we declared our function inside the parenthesis before invoking it.

Function expression

To reuse an anonymous function, we have to assign it to a **variable**. This way, we can reference it by that variable name. A way to do this is through **Function expression**. Let's assign our anonymous function to a variable:

```
const alertUser = function () {
  return alert("I'm anonymous, but with a name!")
}
```

Now, we can invoke (or call) our function the same way we invoked `multiplyByTwo()`. Since our function has no **parameter**, there'll be no need to supply an **argument**.

```
alertUser()
```

Result:

```
I'm anonymous, and you can reference me by a name!
```

Multi-line return statements

> When returning an expression in a function, don't add a new line between `return` and the value.

You might be tempted to separate your expression from the `return` statement like this:

```
function multiplyByFive(value) {
  return
  value * 5
}

console.log(multiplyByFive(10))
// output: undefined
```

Calling the above `multiplyByFive` function will return `undefined`, and this is because our compiler ends the statement after the `return` keyword, making it an empty return. If you want to wrap the return statement across multiple lines, you should start your expression on the same line as the `return` keyword or have an opening parenthesis on the same line as the `return` keyword:

```
function multiplyByFive(value) {
  return value * 5
}

console.log(multiplyByFive(10))
// output: 50
```

Arrow Functions

JavaScript arrow functions, also known as "**fat arrow**" or "**lambda**" functions, are a shorthand way to write function expressions in JavaScript. They were introduced in **ES6** (the 6th major version of JavaScript) and have become popular in modern JavaScript development.

An arrow function is defined using the "fat arrow" syntax, which is a single equals sign followed by a greater than sign (=>). For example, let's use the function declaration syntax you know to write a function that returns the square of a number, that is, the number multiplied by itself:

```
const squareNumber = function (value) {
  return value * value
}
console.log(squareNumber(5))
// Output: 25
```

The equivalent arrow function would look like this:

```
const squareOfNumber = (value) => value * value
```

See that this is a lot shorter than the function declaration syntax. First, we define a variable named `squareOfNumber`, then we write our arrow function as its value. The arrow function takes a parameter named `value` and returns its square. Since the function body is a single expression, we don't need to use curly braces. We can also omit the `return` keyword, and when we do so, the function will return the value of the expression.

If an arrow function has only one parameter, you can omit the parentheses around the parameter list:

```
const squareOfNumber = (value) => value * value
```

You can also write arrow functions with curly braces, but in this case, you must use the `return` keyword:

```
const squareOfNumber = (value) => {
  return value * value
}
```

One of the main benefits of using arrow functions is their conciseness. They are especially useful when you need to write a function that takes another function as a parameter. When a function is used as a parameter to another function, it is called a **callback function**. And the parent function that takes the callback function as a parameter is called a **higher order function**. We'll learn more about these concepts as we progress.

Functions best practices

> The name of a function should be descriptive, straight to the point and timeless.

Notice how we named our first function `multiplyByTwo`. This is because its job is to multiply a number by two. Doing this will help anyone know what our function does by looking at its name.

Function naming is an essential part of **function definition**, and choosing to disregard this will make your code **non-readable**, **non-maintainable**and lead to **reduced efficiency**.

When naming your functions, it is essential not to use slangs that only a few people would understand or trends that people can only relate to at a given time.

Let your function names be descriptive enough, straight to the point and timeless.

> A function should have only one responsibility.

When you define your function, let it have one responsibility and do only what its name suggests. This means that from our examples, choosing to reverse a word and multiply a number in the same function would not be a good idea. Like we did, splitting the two actions into separate functions is better. By doing so, you would not only make your code more readable, but you'll also make it easier to test and debug.

Things to note

- A **function** is a set of statements designed to perform a particular task.
- During a function definition, the names representing the values we intend to supply to the function are called `parameters`.
- During a function call, the actual values provided to the function are called `arguments`.
- We can use up to `255` parameters in a function definition.
- An anonymous function is a function that does not have a name, and to reuse an anonymous function, we have to assign it to a variable.
- An `IIFE` (Immediately Invoked Function Expression) is a function that is invoked immediately after it is declared.
- A function should have only one responsibility.
- Function names should be descriptive, straight to the point and timeless.

JavaScript String Methods

JavaScript has several built-in methods that we can use to manipulate strings. Let's take a look at some of the most common ones.

toUpperCase()

The toUpperCase() method is used to convert a string to uppercase. We can use this method on any string and it'll return a new string with all uppercase characters of the original string Let's see an example:

```
let message = 'Hello World'

console.log(message.toUpperCase())
// output: HELLO WORLD
```

The `toUpperCase()` method does not change the original string. Instead, it returns a new string. So the `message` variable will remain the same:

```
console.log(message)
// output: Hello World
```

toLowerCase()

The toLowerCase() method converts a string to lowercase. Like the `toUpperCase()` method, it does not change the original string. Let's try an example with the `message` variable we created above:

```
console.log(message.toLowerCase())
// output: hello world
```

trim()

The trim() method removes whitespace from the beginning and end of a string:

```
let message = ' Hello World '

console.log(message.trim())
// output: Hello World
```

One important concept to understand is that these methods are chainable, meaning that you can call one method after another on the same string to perform multiple operations in a single line of code. Let's try an example that demonstrates chaining multiple built-in methods to manipulate a string:

```
let message = ' Hello World '

console.log(message.trim().toUpperCase())
// output: HELLO WORLD
```

In the example above, we first call the `trim()` method on the `message` variable to remove the whitespace from the beginning and end of the string. Then we call the `toUpperCase()` method on the result of the `trim()` method to convert the string to uppercase.

Built-in methods in JavaScript provide a lot of functionality for writing readable and concise code. There are some other useful string methods that we'll cover as we progress through this book. It's important to have a good understanding of the methods we've covered so far, and how you can use them to write efficient code.

JS Math functions

JavaScript has a built-in `Math` object that has properties and methods for mathematical constants and operations. Let's take a look at some of the most commonly used methods.

Math.round()

The `Math.round()` method returns the value of a number rounded to the nearest integer. Let's see an example:

```
console.log(Math.round(4.7))
// output: 5
```

Math.ceil()

The `Math.ceil()` method returns the value of a number rounded up to its nearest integer. If a number is already an integer, the method will return the same number:

```
console.log(Math.ceil(4.2))
// output: 5

console.log(Math.ceil(5))
// output: 5
```

The "ceil" in the method name stands for "ceiling", which is a mathematical term for rounding up to the nearest integer.

Math.floor()

The `Math.floor()` method returns the value of a number rounded down to its nearest integer. Like the `Math.ceil()` method, if a number is already an integer, the method will return the same number:

```
console.log(Math.floor(4.7))
// output: 4

console.log(Math.floor(4))
// output: 4
```

The “floor” in the method name is the opposite of “ceiling”.

Math.random()

The `Math.random()` method returns a random number between 0 (inclusive), and 1 (exclusive):

```
console.log(Math.random())
// output: 0.3218232744225875
```

The generated number will be different every time you run the code and is always less than 1. If you want to generate a random number between 0 and 10, you can multiply the result of `Math.random()` by 10:

```
console.log(Math.random() * 10)
// output: 8.688668583064922
```

Math.max()

The `Math.max()` method returns the largest value of a set of numbers. The method expects these numbers to be passed as individual arguments. Let’s see an example:

```
console.log(Math.max(1, 2, 3, 4, 5))
// output: 5
```

Calling `Math.max()` with no arguments will return `-Infinity`:

```
console.log(Math.max())
// output: -Infinity
```

If you pass a non-number argument to `Math.max()`, it will return `NaN` (not a number):

```
console.log(Math.max(1, 2, 3, 4, 5, 'a'))
// output: NaN
```

Math.min()

The `Math.min()` method returns the smallest value of a set of numbers:

```
console.log(Math.min(1, 2, 3, 4, 5))
// output: 1
```

There are a few other useful methods in the `Math` object but these are the most commonly used ones. We can use these methods together to perform complex mathematical operations. For example, since `Math.random()` always returns a decimal number, we can wrap it in the `Math.floor()` method to round it down to the nearest integer so that our generated number is always a whole number:

```
console.log(Math.floor(Math.random() * 10))
// output: 3
```

Arrays

In JavaScript, we use arrays to store multiple values in a single variable. These values can be of any data type, including numbers, strings, and even other arrays. You can also think of an array as a data structure that stores a list of items. Arrays are a fundamental building block of many JavaScript programs. They are used in various contexts, from simple data lists to more complex data structures. Let's learn how to create, access, and manipulate arrays in JavaScript.

Declaring and Initializing Arrays

You can use the Array constructor or the square bracket [] notation to create an empty array. In JavaScript, we use constructors to create data structures with a specific set of properties. Let's create an empty array using the Array constructor. To use a constructor, we need the `new` keyword:

```
1  const myArray = new Array()
```

To create an empty array using the square bracket notation, you can write:

```
1  const myArray = []
```

Both statements create an empty array that you can add elements to later.

You can also initialize an array with elements by passing them as arguments to the Array constructor or using the square bracket notation and separating the elements with commas.

```
const favoriteFruits = new Array(
  'apple',
  'banana',
  'orange'
)
```

or

```
const favoriteFruits = ['apple', 'banana', 'orange']
```

Accessing and modifying Array Elements

To access elements in an array, you can use the square bracket notation, along with the index of the element you want to access. The index of an element is its position in an array, and when locating it, we start counting from 0. So, the first element of an array has an index of 0, the second element an index of 1, the third element an index of 2, and so on. In our `favoriteFruits` array, the first element is 'apple' with an index of 0, the second element is 'banana' with an index of 1, and the third element is 'orange' with an index of 2. Let's use the square bracket notation to access the first element of the `favoriteFruits` array:

```
favoriteFruits[0]
```

This will return the string 'apple' when logged to the console:

```
console.log(favoriteFruits[0])
// output: 'apple'
```

If we want to access the second element, we can write:

```
favoriteFruits[1]
// output: 'banana'
```

You can modify the elements in an array by reassigning the value of a specific index. Let's change the value of the first element in the `favoriteFruits` array to 'strawberry':

```
favoriteFruits[0] = 'strawberry'
```

Now, if you log the first element of the `favoriteFruits` array to the console, you'll get 'strawberry' instead of 'apple':

```
favoriteFruits[0]
// output: 'strawberry'
```

Also, logging the entire array to the console will show the updated value:

```
console.log(favoriteFruits)
// output: ['strawberry', 'banana', 'orange']
```

Array Length

The length property of an array returns the number of elements in it. Let's get the length of the `favoriteFruits` array:

```
console.log(favoriteFruits.length)
// output: 3
```

Unlike Array index, the length of an array is not zero-based. So, when counting the number of elements in an array, we start from 1. In our `favoriteFruits` array, the length is 3, which means that it has 3 elements.

Array Methods

Arrays have several methods that you can use to manipulate and interact with the data stored in them. These methods are built-in functions that are available to every JavaScript array. You can use them to add, remove, rearrange or search for elements in an array. Some of the most common array methods include:

- `push()` - adds one or more elements to the end of an array
- `pop()` - removes the last element from an array
- `shift()` - removes the first element from an array
- `unshift()` - adds one or more elements to the beginning of an array
- `indexOf()` - returns the index of the first occurrence of an element in an array
- `slice()` - returns a shallow copy of a portion of an array into a new array object
- `splice()` - changes the contents of an array by removing or replacing existing elements and/or adding new elements in place
- `join()` - joins all elements of an array into a string

Let's try some examples.

push()

The `push()` method adds one or more elements to the end of an array and returns the new length of the array. Let's add two new fruits to the end of the `favoriteFruits` array:

```
1  favoriteFruits.push('mango', 'pineapple')
2  // output: 5
```

Running the code above will add 'mango' and 'pineapple' to the end of the `favoriteFruits` array, and return its new length, which is 5. If you log the `favoriteFruits` array to the console, you'll see the new elements we just added:

```
console.log(favoriteFruits)
// output: ['strawberry', 'banana', 'orange', 'mango', 'p\
ineapple']
```

pop()

The `pop()` method removes the last element from an array and returns that element. Let's remove the last element from the `favoriteFruits` array:

```
favoriteFruits.pop()
// output: 'pineapple'
```

If you log the `favoriteFruits` array to the console, you'll see that 'pineapple' is no longer there:

```
console.log(favoriteFruits)
// output: ['strawberry', 'banana', 'orange', 'mango']
```

shift()

This method removes the first element from an array and returns the removed element:

```
favoriteFruits.shift()
// output: 'strawberry'
```

unshift()

The `unshift()` method adds one or more elements to the front of an array and returns its new length:

```
favoriteFruits.unshift('lemon', 'grape')
// output: 5
```

Logging the `favoriteFruits` array to the console will show the new elements we just added:

```
console.log(favoriteFruits)
// output: ['lemon', 'grape', 'banana', 'orange', 'mango']
```

indexOf()

You can use the `indexOf()` method to find the index of the first occurrence of an element in an array. Let's find the index of 'banana' in the `favoriteFruits` array:

```
favoriteFruits.indexOf('banana')
// output: 2
```

This will return `2` because 'banana' is the third element in the `favoriteFruits` array, which means that it has an index of `2`. Remember that we start counting from `0` when identifying the index of an array element. If the element you're looking for is not in the array, the `indexOf()` method will return `-1`:

```
favoriteFruits.indexOf('watermelon')
// output: -1
```

slice()

The `slice()` method is used to extract a portion of an array. The method takes two arguments: the starting index (inclusive) and the ending index (exclusive) of the slice. The element at the starting index will be included in the slice, but the element at the ending index will not. The `slice()` method does not modify the original array. Instead, it returns a new array containing the extracted elements. Let's extract the first three elements from the `favoriteFruits` array and put them in a new array called `topThreeFruits`:

```
const topThreeFruits = favoriteFruits.slice(0, 3)
console.log(topThreeFruits)
// output: ['lemon', 'grape', 'banana']
```

If you don't specify the ending index, the slice will go all the way to the end of the array:

```
const fruitsInTheBasket = favoriteFruits.slice(2)
console.log(fruitsInTheBasket)
// output: ['banana', 'orange', 'mango']
```

Remember that the value of `favoriteFruits` is still `['lemon', 'grape', 'banana', 'orange', 'mango']`. In the code above, our slice started at index 2, which is 'banana', and since we didn't specify an ending index, the slice went all the way to the end of the array, giving us `['banana', 'orange', 'mango']`.

The `slice()` method can also take a negative index. If you pass a negative index, the slice will start from the end of the array. So to get the last element of the `favoriteFruits` array, we can write:

```
const lastFruit = favoriteFruits.slice(-1)
console.log(lastFruit)
```

If we want to get the last two elements of the array, we can write:

```
const lastTwoFruits = favoriteFruits.slice(-2)
console.log(lastTwoFruits)
// output: ['orange', 'mango']
```

When using a negative index, keep in mind that it represents the element counting from the end of an array, so it might get confusing if you're not familiar with the array length.

splice()

The `splice()` method changes the contents of an array by removing or replacing existing elements. It can take more than two arguments; the first is the starting index, the second is the number of elements to remove, and the rest are the elements to add to the array. Unlike the `slice()` method, The `splice()` method modifies the original array. Let's replace the first and second elements of the `favoriteFruits` array with 'cherry' and 'avocado':

```
favoriteFruits.splice(0, 2, 'cherry', 'avocado')
console.log(favoriteFruits)
// output: ['cherry', 'avocado', 'banana', 'orange', 'man\
go']
```

If you want to remove elements from the array without adding any new elements, you can omit the third argument:

```
favoriteFruits.splice(0, 2)
console.log(favoriteFruits)
// output: ['banana', 'orange', 'mango']
```

join()

The `join()` method joins all elements of an array into a string. It takes an optional argument, which is the separator to use between

the elements. If you don't specify a separator, the elements will be joined with a comma. The `join()` method does not modify the original array, so if you want to join the elements our `favoriteFruits` array and store the result in a new variable, you can write:

```
const favoriteFruitsString = favoriteFruits.join()
console.log(favoriteFruitsString)
// output: 'banana,orange,mango'
```

You can pass a different separator as an argument:

```
const newFavoriteFruitsString =
  favoriteFruits.join(' and ')

console.log(newFavoriteFruitsString)
// output: 'banana and orange and mango'
```

reverse()

The `reverse()` method reverses the order of the elements in an array. Let's reverse the order of the elements in the `favoriteFruits` array:

```
favoriteFruits.reverse()
console.log(favoriteFruits)
// output: ['mango', 'orange', 'banana']
```

There are other commonly used methods you can use to manipulate arrays, including **Higher Order Functions (HOFs)**, which we'll cover in the next section.

Arrays are one of the most important data structures in JavaScript, and understanding how to work with them is an essential aspect of becoming a proficient developer. Remember that you must practice

a lot to become comfortable with any JavaScript concept, so don't hesitate to try out new examples and experiment with the methods we've covered.

Higher Order Functions (HOFs) and Callbacks

We've seen how to create functions that take strings and numbers as arguments. Sometimes, we'll need to create functions that take other functions as arguments or use them as their return values. These are called **higher-order functions**. They are commonly used in JavaScript to manipulate arrays and simplify complex tasks. When a function is passed as an argument to another function, it is called a **callback function** (to be called back at a later time). A callback function can be passed to another function as an argument, returned by another function, and assigned as a value to a variable. When working with JavaScript, you'll often use higher-order functions and callback functions to solve different kinds of tasks.

There are several built-in higher-order functions that we can use to manipulate arrays. For example, every array has a **`map()`** function we can use to modify each element in it. The `map()` function is a higher-order function that takes a callback function as an argument. The callback function is applied to each element in the array, then the `map()` function returns a new array with the results of the callback function applied to each element in the original array. Here's an example of using `map()` to square each element in an array:

```
const numbers = [1, 2, 3, 4, 5]
const squaredNumbers = numbers.map(
  (number) => number * number
)

console.log(squaredNumbers)
// Output: [1, 4, 9, 16, 25]
```

In this example, the anonymous function `(number => number * number)` is passed as an argument to the `map()` function. In the callback function, the `number` parameter represents each element in the `numbers` array. This can be any name we want, but it's common to use the singular form of the array name.

The `map()` function applies the callback function to each element in the array and returns a new array named `sauaredNumbers` with the square of each element in the original array.

Let's try another example using the `map()` function to capitalize each element in an array of strings:

```
const words = ['this', 'is', 'such', 'a', 'great', 'day']
const capitalizedWords = words.map((word) =>
  word.toUpperCase()
)

console.log(capitalizedWords)
// Output: ['THIS', 'IS', 'SUCH', 'A', 'GREAT', 'DAY']
```

In this example, we use the `map()` function to capitalize each element in the `words` array. The `map()` function does not modify the original array, so we have to assign its return value to a new array named `capitalizedWords`.

Creating Higher-order functions

We can create our higher-order functions ourselves. All we need to do is add a function parameter or return another function when it is called. Here's an example of a higher-order function that takes a callback function as an argument:

```
const higherOrderFunction = (callback) => {
  const string = 'HOFs are really cool!'
  callback(string)
}

higherOrderFunction(console.log)

// Output: 'HOFs are really cool!'
```

See that our `higherOrderFunction()` takes a callback function as an argument, then calls that function with the string 'HOFs are really cool!'. In this case, supplying `console.log()` as an argument to the `higherOrderFunction()` logs the string to the console.

We can also return a function from another function. Here's an example of a higher-order function that returns a callback function:

```
const callMeTwice = () => console.log

callMeTwice()('Hey, this works!')
// Output: 'Hey, this works!'
```

In this example, the `callMeTwice()` function returns the `console.log()` function. So, when we call `callMeTwice()`, what we get is the `console.log()` function. So, we'll have to call the result of `callMeTwice()` to use the `console.log()` function. Chaining function calls like this is known as **currying**. In currying, each function call returns another function that can be called again

with a different argument. The main goal is to simplify the process of passing arguments to a function by breaking it down into a series of function calls, each taking a single argument.

Built-in HOFs and Array Methods

Let's explore other built-in higher-order functions you can use to manipulate arrays.

forEach()

The `forEach()` function is an array method that executes a callback function on each of its elements. Unlike the `map()` function, the `forEach()` function does not return a new array and does not modify the original array either. Instead, it iterates through the elements of an array and performs an operation on each of them. **To iterate** means to repeat a specific action or set of actions a number of times or until a condition is met.

We can use the `forEach()` function to log each element in an array to the console or perform an action with each of them and push the results to a new array. Let's use the `forEach()` function to log each element in an array to the console:

```
1  const namesOfPeople = ['oluchi', 'jude', 'kanye', 'tiwa']
2
3  namesOfPeople.forEach((name) => console.log(name))
```

This will output:

```
oluchi
jude
kanye
tiwa
```

Notice that the names of people in our array start with lowercase letters. Here's how we can title-case each element in the array and push the results to a new array. Title-casing means capitalizing the first letter of each word in a string:

```
const titleCasedNames = []

namesOfPeople.forEach((name) => {
  const titleCasedName =
    name[0].toUpperCase() + name.slice(1)
  titleCasedNames.push(titleCasedName)
})

console.log(titleCasedNames)
// Output: ['Oluchi', 'Jude', 'Kanye', 'Tiwa']
```

In this example, we use the `forEach()` function to iterate through the `namesOfPeople` array. For each string in the array, we capitalize its first letter using the square bracket notation and the `toUpperCase()` method. *When working with strings, we can access each character in the string using the square bracket notation and the index of the character.* The `slice()` method returns a new string with the characters from the index we specify to the end of the string. We then push the title-cased name to the `titleCasedNames` array.

filter()

The `filter()` function is another array method we can use to filter out elements that meet a condition. For example, if we have an

array of countries, we can use the `filter()` function to filter out countries that start with the letter 'N':

```
const countries = [
  'Nigeria',
  'UK',
  'Netherlands',
  'Canada',
  'USA',
  'Norway',
]

const countriesThatStartWithN = countries.filter(
  (country) => country.startsWith('N')
)

console.log(countriesThatStartWithN)
// Output: ['Nigeria', 'Netherlands', 'Norway']
```

In JavaScript, every string has a `startsWith()` method that returns `true` if the string starts with the character we specify. In our example, the filter method takes a callback function with a parameter named `country` representing each element in the `countries` array. The callback function returns `true` if the country starts with the letter 'N', and the `filter()` function returns a new array with the elements that meet the condition.

We can also use the square bracket notation or the `charAt()` method to check if a string starts with a character. The `charAt()` method returns the character at the specified index in a string:

```
const countriesThatStartWithU = countries.filter(
  (country) => country.charAt(0) === 'U'
)

console.log(countriesThatStartWithU)
// Output: ['UK', 'USA']
```

We can use the `filter()` function to filter out elements that don't meet a condition. Let's filter out countries that don't start with the letter 'N':

```
const countriesThatDontStartWithN = countries.filter(
  (country) => !country.startsWith('N')
)

console.log(countriesThatDontStartWithN)
// Output: ['UK', 'Canada', 'USA']
```

In this example, we use the logical NOT operator (`!`) to negate the condition so that the `filter()` function returns a new array with elements that don't meet the condition. When using the filter method, it is important to remember that the callback function should return a boolean value.

sort()

The `sort()` method is used to sort the elements of an array. The `sort()` method modifies the original array and returns the sorted array. For example, if we want to sort an array of numbers, we can use the `sort()` method. By default, the `sort()` method sorts the elements of an array in ascending order:

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

numbers.sort()

console.log(numbers)
// Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

We can also sort an array of strings:

```
const favoriteColors = [
  'red',
  'blue',
  'green',
  'yellow',
  'orange',
]

favoriteColors.sort()

console.log(favoriteColors)
// Output: ['blue', 'green', 'orange', 'red', 'yellow']
```

See that blue comes first because it starts with the letter 'b' which comes before the letter 'g' for green, and so on. If we want to sort the elements of an array in descending order, we can pass a callback function to the `sort()` method that takes two parameters representing two elements in the array. The callback function should return a negative number if the first element should come before the second element, a positive number if the first element should come after the second element, or `0` if the order of the elements doesn't matter:

```
const famousBirthYears = [
  1990, 2001, 1989, 2023, 2005, 1993, 2020,
]

famousBirthYears.sort((a, b) => b - a)

console.log(famousBirthYears)
// Output: [2023, 2020, 2005, 2001, 1993, 1990, 1989]
```

In this example, we sort the `famousBirthYears` array in descending order. The callback function takes two parameters, `a` and `b`, representing two elements in the array. Since we want to sort the array in descending order, we return `b - a` so that the second element comes before the first element (i.e., the bigger number comes before the smaller number). If we wanted to sort the array in ascending order, we would return `a - b` so that the first element comes before the second element (i.e., the smaller number comes before the bigger number):

```
famousBirthYears.sort((a, b) => a - b)

console.log(famousBirthYears)
// Output: [1989, 1990, 1993, 2001, 2005, 2020, 2023]
```

reverse()

We can use the `reverse()` method to reverse the order of the elements in an array. The reverse method modifies the original array. Let's see an example:

```
const agesOfStudents = [13, 10, 15, 12, 11, 14]

agesOfStudents.reverse()

console.log(agesOfStudents)
// Output: [14, 11, 12, 15, 10, 13]
```

Notice that the `reverse()` method does not sort the elements of an array. Instead, it reverses the order of the elements from the last to the first.

reduce()

The `reduce()` method reduces an array to a single value. It iterates through an array, performs an action on each element, and returns a single value. For example, if we want to add all the elements in an array of numbers, we can do so with the `reduce()` method:

```
const participantPoints = [10, 20, 30, 40, 50]

const totalPoints = participantPoints.reduce(
  (a, b) => a + b
)

console.log(totalPoints)
// Output: 150
```

The `reduce()` method takes a callback function with two parameters, `a` and `b`, representing two elements in the array. The callback function then performs an action starting from the first element in the array. In our example, we add the first element to the second element, then add the result to the third element, and so on, until we get the total points. If we want to minus the elements in the array, we would return `a - b`:

```
const minusedPoints = participantPoints.reduce(
  (a, b) => a - b
)

console.log(totalPoints)
// Output: -130
```

Our example above performs the action; `10 - 20 - 30 - 40 - 50`. We can use the same process to multiply the elements in the array:

```
const multipliedPoints = participantPoints.reduce(
  (a, b) => a * b
)

console.log(multipliedPoints)
// Output: 12000000
```

This time, we multiply the first element by the second element, then multiply the result by the third element, and so on. The `reduce()` method does not modify the original array.

We can also use the `reduce()` method to concatenate the elements in an array of strings:

```
const favoriteDishes = [
  'jollof',
  'burger',
  'yam',
  'meatloaf',
  'pizza',
]

const favoriteDishesString = favoriteDishes.reduce(
  (a, b) => a + ', ' + b
)
```

```
12
13 console.log(favoriteDishesString)
14 // Output: jollof, burger, yam, meatloaf, pizza
```

Without adding the comma and space, the output would be `jollofburgeryammeatloafpizza`. Sounds like I just invented a new dish. Haha!

every()

The `every()` method checks if all the elements in an array meet a condition. It returns a boolean value. For example, if we want to check if all the elements in an array of numbers are greater than 5, we can do so with the `every()` method:

```
1 const participantAges = [20, 25, 14, 29, 30, 17]
2
3 const areAllParticipantsOlderThan18 =
4   participantAges.every((age) => age > 18)
5
6 console.log(areAllNumbersGreaterThan5)
7 // Output: false
```

This example checks if all the elements in the `participantAges` array are greater than 18. Since at least one element is less than 18, the `every()` method returns `false`. For the `every()` method to return `true`, all the elements in the array must meet the condition. Let's check if all the elements in the `participantAges` array are greater than 10:

```
const areAllParticipantsOlderThan10 =
  participantAges.every((age) => age > 10)

console.log(areAllParticipantsOlderThan10)
// Output: true
```

This time, all the elements in the array are greater than 10, so the `every()` method returns `true`.

some() and includes()

The `some()` method is similar to the `every()` method, but unlike the `every()` method, it checks if at least one element in an array meets a condition, and then returns a boolean value. Let's check if at least one element in an array of cities has the letter 'e' in it:

```
const cities = [
  'Massachusetts',
  'Enugu',
  'Vienna',
  'Abuja',
  'London',
]

const atLeastOneCityHasE = cities.some((city) =>
  city.includes('e')
)

console.log(atLeastOneCityHasE)
// Output: true
```

In JavaScript, the `includes()` method checks if a string contains a specified substring, or if an array contains a specified element. In our example, the `some()` method returns `true` because at least one

element in the `cities` array has the letter 'e' in it. If we used the `every()` method instead, it would return `false` because not every element has the letter 'e' in it.

Let's use the `includes()` method to check if the cities array contains the city 'Abuja':

```
console.log(cities.includes('Abuja'))
// Output: true
```

find()

The `find()` method returns the first element in an array that meets a condition. For example, if we want to find the first element in an array of quantities that is greater than 10, we can use the `find()` method to do so:

```
const quantities = [7, 3, 9, 13, 5, 11]

const firstQuantityGreaterThan10 = quantities.find(
  (quantity) => quantity > 10
)

console.log(firstQuantityGreaterThan10)
// Output: 13
```

Notice that even though 11 is also greater than 10, the `find()` method only returns 13 because it is the first element that meets the condition. If no element in the array meets the condition, the `find()` method returns `undefined`.

findIndex()

We can use the `findIndex()` method to find the index of the first element in an array that meets a condition. Let's find the index of

the first element in an array of precious metals that starts with the letter 'p':

```
const carColor = 'red'
const carMake = 'Tesla'
const carModel = 'Cybertruck'
const carMileage = 30
```

The `findIndex()` method returns the index of the first element that starts with 'p'. If no element in the array meets the condition, the `findIndex()` method will return `-1`.

JavaScript array methods are instrumental in working with and manipulating arrays. Without them, we would have to write a lot of code to perform the same actions. It's important to understand how to use these array methods so that you can write more efficient code that is easy to understand and maintain. Remember that to master JavaScript, you must practice and experiment with a lot of examples.

The JavaScript Timing methods

The JavaScript timing methods are built-in higher-order functions that allow us to schedule the execution of functions at a later time or at regular intervals. Let's take a look at the most commonly used timing methods.

setTimeout()

The `setTimeout()` method allows us to schedule the execution of a function after a specified number of milliseconds. For example, if we want to display a message after 5 seconds, we can use the `setTimeout()` method to do so. The `setTimeout()` method takes

two arguments: the function to be executed and the number of milliseconds to wait before executing the function. Let's display a message after 5 seconds:

```
const displayMessage = () => {
  console.log('Hello, world!')
}

setTimeout(displayMessage, 5000)
// Output: 2
// Output: Hello world!
```

1 second is a thousand milliseconds and 5 seconds is 5000 milliseconds. The `setTimeout()` method will wait for 5 seconds before executing the `displayMessage()` function. The `setTimeout()` method also returns a number that represents the ID of the timer immediately after it is called. We can use this ID to cancel the timer. First, let's assign the setTimeout method to a variable:

```
const displayMessage = () => {
  console.log('Hello, world!')
}

const timerId = setTimeout(displayMessage, 5000)

console.log(timerId)
// Output: 4 (can be any number)
// Output: Hello world!
```

This will still display the message after 5 seconds, but now we have the ID of the timer in the `timerId` variable. Let's use it to cancel the timer before it executes the `displayMessage()` function. We'll use the `clearTimeout()` method to do so. The `clearTimeout()` method takes the ID of the timer as an argument and cancels the timer:

```
const displayMessage = () => {
  console.log('Hello, world!')
}

const timerId = setTimeout(displayMessage, 5000)

clearTimeout(timerId)
```

The `clearTimeout()` method will cancel the timer before it executes the `displayMessage()` function. So when you run this code, the `displayMessage()` function will not be executed.

setInterval()

The `setInterval()` method allows us to schedule the execution of a function at regular intervals. For example, if we want to display our message every 5 seconds, we can use the `setInterval()` method to do so. The `setInterval()` method takes two arguments: the function to be executed and the number of milliseconds to wait before executing the function again. We can directly call the method or assign it to a variable. Like the `setTimeout()` method, the `setInterval()` method also returns a number that represents the ID of the timer. Let's use the `setInterval()` method to run the `displayMessage()` function every 1 second:

```
const displayMessage = () => {
  console.log('Hello, world!')
}

const interVal = setInterval(displayMessage, 1000)
```

This will display the message "Hello world!" every 1 second. We can use the `clearInterval()` method to cancel the timer. The `clearInterval()` method takes the ID of the timer as an argument and cancels the timer:

```
clearInterval(interVal)
```

Running the `clearInterval()` method will stop the `displayMessage()` function from being executed every second.

JavaScript Objects

Imagine you have a car and want to keep track of its color, make, model and mileage. You could create a variable for each of these properties:

```
const carColor = 'red'
const carMake = 'Tesla'
const carModel = 'Cybertruck'
const carMileage = 30
```

But what if you had 10 cars? You would have to create 4 variables for each car, giving you 40 variables in total. And what if you wanted to add a new property? Like the type of fuel the car uses? You would have to create ten new variables. As you keep creating variables, it'll become harder to know which ones are related to each other or which cars they belong to.

With JavaScript objects, you can group related data in a single variable. So instead of creating four different variables for your first car, you can create one that stores all four properties. And if you want to add a new property, you can do so without creating a new variable. Here's what using an object to store your car's data would look like:

```
const myCar = {
  color: 'red',
  make: 'Tesla',
  model: 'Cybertruck',
  mileage: 30,
}
```

In the example above, we created a variable named `myCar` and assigned it an object. You'll recognize objects by the curly braces surrounding them, followed by a list of key-value pairs. Each key-value pair is a property of the object. In our example, the first property is the color of the car. Its key is `color` and value, `"red"`. Unlike a variable declaration, the key and value are separated by a colon, and a comma separates each property.

To access a property of an object, you can use dot notation or bracket notation. Dot notation is the most common way to access properties; first, you type the name of the object, followed by a period, and then the name of the property. Here's how you would use dot notation to access the value of the `color` property:

```
console.log(myCar.color)
// output: "red"
```

And here's how you would use bracket notation:

```
console.log(myCar['color'])
```

Bracket notation is useful when you want to access a property whose name is stored in a variable. For example, let's create a variable named `propertyName` and assign it the string `"color"`:

```
const propertyName = 'color'

console.log(myCar[propertyName])
// output: "red"
```

Rules and Guidelines for creating objects

When creating objects, there are a few things to keep in mind:

Property names of an object must be unique

Multiple properties can have the same value, but If you try to create a property with the same name as an existing property, the new property will override the old one. Also, if an object has multiple properties with the same name, only the last one will be used.

```
const myFavoriteBook = {
  title: 'The Hobbit',
  title: 'The Lord of the Rings',
}
console.log(myFavoriteBook.title)
// Output: "The Lord of the Rings"
```

Property names must be strings

When you use a non-string as a property name, it will be converted to a string. For example, the following code will create a property named `"1"`:

```
const myFavoriteDress = {
    1: "green"
}

console.log(myFavoriteDress.1)
// Output: SyntaxError
```

Trying to access the key `1` using dot notation will result in a syntax error because the number `1` is not a valid key, and has been converted to the string `"1"`. In this case, you can use bracket notation to access it:

```
console.log(myFavoriteDress['1'])
// Output: "value1"
```

invalid JavaScript identifiers must be enclosed in quotes

Object keys can be any string, including an empty string. However, if the key is a string that is not a valid JavaScript identifier, it must be enclosed in quotes. For example, adding spaces to a key will result in a syntax error:

```
const myFavoriteMeal = {
    plate type: "ceramic"
}
// Output: SyntaxError
```

But enclosing the key in quotes will make it a valid property name:

```
const myFavoriteMeal = {
    "plate type": "ceramic"
    "": "valid but not recommended"
}

console.log(myFavoriteMeal["plate type"])
// Output: "ceramic"

console.log(myFavoriteMeal[""])
// Output: "valid but not recommended"
```

Notice that we used bracket notation to access the property. You can't use the dot notation when the key contains spaces or other invalid characters:

```
const myFavoriteCar = {
  'engine*type': 'electric',
}

console.log(myFavoriteCar.engine * type)
// Output: NaN

console.log(myFavoriteCar['engine*type'])
// Output: "electric"
```

Property names are case-sensitive

Remember that JavaScript is a case-sensitive language. This also applies to object properties. For example, the following code will create two properties, one named `"title"` and one named `"Title"`:

```
const myFavoriteMovie = {
  title: 'Titanic',
  Title: 'The Lion King',
}

console.log(myFavoriteMovie.title)
// Output: "Titanic"

console.log(myFavoriteMovie.Title)
// Output: "The Lion King"
```

Like with variables, it's a good practice to use camelCase for property names to make your code style consistent and easy to read.

Property values can be any valid JavaScript expression

The value of a property can be any valid JavaScript expression, including a function. When a function is used as a property value, it's called a method, which is why we've been using the term "method" when referring to inbuilt JavaScript functions like `console.log()`. Let's create an object with a method named `sayHello`:

```
const person = {
  fullName: 'Victor Otedola',
  sayHello: function () {
    console.log('Hello! My name is ' + this.name)
  },
}

person.sayHello()
// Output: Hello! My name is Victor Otedola
```

In the example above, we created an object named `person` with a `fullName` property and a method named `sayHello`. The `sayHello` method uses the **`this` keyword** to access the `name` property of the object. When you use the `this` keyword inside a method, it refers to the object on which the method is being called. In this case, `this` refers to the `person` object.

You can also use an arrow function as a method, but the `this` keyword will not work the same way, instead, its value will be determined by the context in which the method is called:

```
const person2 = {
  fullName: 'Linda Chima',
  sayHello: () => {
    console.log('Hello! My name is ' + this.name)
  },
}

person2.sayHello()
// Output: Hello! My name is undefined
```

The `this` keyword doesn't work the same way as the first example because it is used in an arrow function. In this case, `this` refers to the global object, which doesn't have a `name` property. This can get confusing, so it's best to avoid using `this` in arrow functions.

Adding, modifying and Removing Properties

You can add new properties to an object using dot or bracket notation. Let's create a new object named `student` and add a new property, `gender`:

```
const student = {
  name: 'Kachi',
  age: 17,
}

student.gender = 'Male'

console.log(student)
// Output: { name: "Kachi", age: 17, gender: "Male" }
```

Here's how to use bracket notation to add a new property:

```
student['grade'] = 12

console.log(student)
// Output: { name: "Kachi", age: 17, gender: "Male", grad\
e: 12 }
```

You can also use the dot and bracket notation to modify an existing property:

```
student.age = 18
student['grade'] = 13

console.log(student.age) // Output: 18
console.log(student.grade) // Output: 13
```

JavaScript has an `Object.defineProperty()` method for adding a new property to an object. The `Object.defineProperty()` method is a way to define or modify a property on an object. It is used to control the behavior of a property and make it read-only (cannot be modified), non-enumerable (cannot be iterated over), or non-configurable (cannot be deleted or modified). Here's how to use it to add a new property to an object:

```
Object.defineProperty(student, 'school', {
  value: 'University of Lagos',
  writable: false,
  enumerable: true,
  configurable: true,
})

console.log(student)
```

Output:

```
{
name: "Kachi",
age: 18,
gender: "Male",
grade: 13,
school: "University of Lagos"
}
```

The first argument is the object on which to add the property. The second argument is the name of the property. The third argument is an object that contains the property's attributes. The `value` attribute is required, and it specifies the value of the property. The `writable`, `enumerable`, and `configurable` attributes are optional. They will default to `false` if you don't specify them. Notice that we set the `writable` attribute of the `school` property to `false`, so changing its value won't work:

```
student.school = 'Havard University'

console.log(student.school)
// Output: "University of Lagos"
```

To remove a property, you can use the `delete` keyword:

```
delete student.school

console.log(student.school)
// Output: undefined
```

Let's try using `Object.defineProperty()` to add a new property to the student object that can't be deleted:

```
Object.defineProperty(student, 'language', {
  value: 'English',
  writable: true,
  enumerable: true,
  configurable: false,
})

console.log(student.language)
// Output: "English"

delete student.language

console.log(student.language)
// Output: "English"
```

The `configurable` attribute is set to `false`, so the `language` property can't be deleted.

Object.prototype

In JavaScript, every object has a prototype. When you create an object, it inherits a prototype, which is an object that contains other built-in properties and methods you can use on the main object. For example, every object has a `toString()` method that returns a string representation of the object, even though you don't add it yourself:

```
const computerInfo = {
  brand: 'Apple',
  model: 'MacBook Pro',
  color: 'Silver',
}

console.log(computerInfo.toString())

// Output: [object Object]
```

The default string representation of an object is `[object Object]`. If you want to get a string representation of an object that contains the object's properties, you can use the built-in `JSON.stringify()` method:

```
JSON.stringify(computerInfo)

// Output: '{"brand":"Apple","model":"MacBook Pro",
// "color":"Silver"}'
```

To access all the properties and methods of an object's prototype, you can use the built-in `Object.getPrototypeOf()` method:

```
Object.getPrototypeOf(computerInfo)

// Output: {propertyIsEnumerable: f, toString: f, ...}
```

The `Object.getPrototypeOf()` method returns an object that contains all the properties and methods of the object's prototype. You can also use it to add new properties and methods to an object's prototype:

```
Object.getPrototypeOf(computerInfo).isExpensive = true

console.log(computerInfo.isExpensive)
// Output: true
```

Object methods

There are other methods available on the built-in `Object` object that you can use to add, modify and retrieve data from any object. You've already seen the `Object.defineProperty()` method. Let's explore some other useful methods.

Object.keys()

The `Object.keys()` method returns an array of a given object's enumerable property names. Let's get the keys of the `computerInfo` object:

```
Object.keys(computerInfo)

// Output: ["brand", "model", "color"]
```

Object.values()

The `Object.values()` method returns an array of a given object's enumerable property values:

```
Object.values(computerInfo)

// Output: ["Apple", "MacBook Pro", "Silver"]
```

Object.entries()

The `Object.entries()` method returns an array of a given object's enumerable property `[key, value]` pairs:

```
Object.entries(computerInfo)

// Output: [["brand", "Apple"], ["model", "MacBook Pro"],
// ["color", "Silver"]]
```

In the return value, each element is an array containing the property name and value.

Object.assign()

The `Object.assign()` method copies all enumerable own properties from one or more source objects to a target object. It returns the modified target object. Let's create a new object named `newComputerInfo` and copy the properties of the `computerInfo` object to it:

```
const newComputerInfo = Object.assign({}, computerInfo)

console.log(newComputerInfo)
// Output: {brand: "Apple", model: "MacBook Pro",
// color: "Silver"}
```

Object.freeze()

The `Object.freeze()` method freezes an object so it can't be modified. Let's freeze the `computerInfo` object and try to modify it:

```
Object.freeze(computerInfo)

computerInfo.brand = 'Dell'

console.log(computerInfo.brand)
// Output: "Apple"
```

The `brand` property of the `computerInfo` object can't be modified because it's frozen.

Object.seal()

The `Object.seal()` method seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable. Let's seal the `computerInfo` object and try to add a new property to it:

```
Object.seal(computerInfo)

computerInfo.price = 200000

console.log(computerInfo.price)
// Output: undefined
```

These are the most common methods in the `Object` object, but you can find more in the MDN documentation*.

Understanding how to use these methods is essential for mastering objects in JavaScript. So, make sure you keep practicing until you're comfortable with them.

You'll find yourself using objects a lot. They are an important part of JavaScript, and it's almost impossible to build applications without grouping related data in objects. They help make your code organized, easy to read, and maintainable.

*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

Object destructuring

Object destructuring is another way to access the properties of an object. With destructuring, we can extract values from an object and assign them to variables with the same name as the object's properties. It is done by enclosing the properties in curly braces `{}` on the left-hand side of the assignment operator `=`. Let's see an example. First, we'll create an object that stores information about an attendee at a conference:

```
const attendee = {
  name: 'Peter Okoye',
  age: 30,
  email: 'peterokoye@nonexistentmail.com',
  ticketType: 'VIP',
}
```

Now, let's access the properties of the `attendee` object using destructuring:

```
const { name, age, email, ticketType } = attendee

console.log(name)
// Output: "Peter Okoye"

console.log(age)
// Output: 30

console.log(email)
// Output: "peterokoye@nonexistentmail.com"

console.log(ticketType)
// Output: "VIP"
```

We don't have to destructure all the properties of the object at once. We can destructure only the properties we need:

```
const { name } = attendee
```

We can also assign default values to the variables in case the object doesn't have the property:

```
const { name, isPaid = false } = attendee

console.log(isPaid)
// Output: false
```

We can assign a different name to the variable by using the colunm : and the new name after the property name. This is useful when we want to avoid naming conflicts:

```
const { name: attendeeName } = attendee

console.log(attendeeName)
// Output: "Peter Okoye"
```

Destructuring arrays

We can also similarly destructure arrays. Let's create an array that stores the names of the attendees at a conference:

```
const attendeeNames = [
  'Peter Okoye',
  'Paul Okoye',
  'Lynda Okoye',
  'Rita Okoye',
]
```

To destructure the array, we enclose the elements in square brackets `[]` on the left-hand side of the assignment operator `=`. Since the elements of an array are not named, we can only destructure them in the order they appear in the array. So the first element of the array will be assigned to the first variable, the second element to the second variable, and so on. Let's destructure the `attendeeNames` array:

```
const [
  firstAttendee,
  secondAttendee,
  thirdAttendee,
  fourthAttendee,
] = attendeeNames

console.log(firstAttendee)
// Output: "Peter Okoye"

console.log(thirdAttendee)
// Output: "Lynda Okoye"
```

We can also destructure only the elements we need by skipping the elements we don't need with commas `,`. Let's skip the second element of the array:

```
const [firstAttendee, , thirdAttendee] = attendeeNames
```

Object and array destructuring can make code more readable and concise by simplifying the process of extracting values from

objects and arrays, and assigning them to variables. That being said, whether or not to use object and array destructuring depends on the specific use case and personal preference. In some cases, it might be unnecessary, but in most cases where you find yourself reusing object properties or assigning them to variables, it's a good idea to consider destructuring.

Conditional Statements

We've learned about basic conditional statements and how to use the `if-else` statement to execute code based on a condition. Conditional statements are used to control the flow of a program, just like you live your everyday life. You make decisions based on your circumstances and execute different actions accordingly. For example, if you're cold, you might wear a jacket; if it's raining, you might wear a raincoat instead; and if it's sunny, you might wear a t-shirt. You can use the basic `if-else` statement to handle the first condition (if it's cold), but when you need to check multiple conditions, you'll need more complex conditional statements like the `else-if` and `switch` statements.

Else-if statements in JavaScript

To check multiple conditions, you can use an else-if clause. The else-if clause allows you to chain several conditional statements together so that if the first condition is not met, the second condition is checked, and so on. Here's a simple example to illustrate how the else-if clause works:

```
if it's cold: wear a jacket
else if it's raining: wear a raincoat
else: wear a t-shirt
```

Let's see an example in JavaScript. We'll use the `console.log()` method to display text on the console if a condition is met:

```
if (typeof 'Hello' === 'number') {
  console.log('The first condition was met!')
} else if (typeof 10 === 'number') {
  console.log('The second condition was met!')
} else {
  console.log('None of the conditions were met.')
}

// output: The second condition was met!
```

Since the first condition `typeof('Hello') === 'number'` is not met, the following else-if statement is evaluated. In our example, the second condition `(typeof(10) === 'number')` is true, so the code inside the else-if clause is executed. If the second condition was not met, the code inside the next `else` clause would be executed instead. An else-if clause is only evaluated if the preceding condition is not met. Let's change the first condition to `typeof('Hello') === 'string'` and run the code again:

```
if (typeof 'Hello' === 'string') {
  console.log('The first condition was met!')
} else if (typeof 10 === 'number') {
  console.log('The second condition was met!')
} else {
  console.log('None of the conditions were met.')
}

// output: The first condition was met!
```

When you run the above code on your console, you'll get the string "The first condition was met!" and the following `else-if` clause will not be evaluated. The `else-if` clause is only evaluated if its preceding condition is not met. If you want to evaluate the second condition regardless of the output of the first, you should use an `if` statement instead. Here's an example:

```
if (typeof 'Hello' === 'string') {
  console.log('The first condition was met!')
}
if (typeof 10 === 'number') {
  console.log('The second condition was met!')
} else {
  console.log('None of the conditions were met.')
}

// output: The first condition was met!
// output: The second condition was met!
```

When you run the above code, both strings will be logged to the console because the second condition is also an `if` statement. You should only use the `else-if` clause when writing connected conditions. Notice that we only have one `else` clause in our examples. The `else` clause is executed if none of the conditions are met. It also indicates the end of the conditional statement. So if you write an `else` clause and try to add an `else-if` clause after it, you'll get an error:

```
if (typeof('Hello') === 'string') {
  console.log('The first condition was met!')
} else {
  console.log('The condition was not met')
} else if (typeof(10) === 'number') {
  console.log('The second condition was met!')
}

// output: SyntaxError: Unexpected token else
```

You can have as many else-if clauses as you want in an if statement, but you can only have one `else` clause.

Nested if statements

Sometimes, you'll need to check multiple conditions inside a single condition. For example, if your laptop works, but the battery is low, you might want to charge it. If the laptop works and the battery is 50% or more, you might want to use it. If the laptop works, but the internet connection is unavailable, you might want to restart the router. You can use regular if-else statements to check all these conditions:

```
if laptop works but battery is low: charge it
else if laptop works but no internet: restart the router
else if laptop works and battery is 50% or more: use it
```

Notice how every condition is dependent on one; "If the laptop works". Writing this one condition multiple times is redundant. Instead, we can write it once and nest the other conditions inside it. Here's how our simple illustration would look like using nested if statements:

```
if laptop works:
  if battery is low: charge it
  else if no internet: restart the router
  else if battery is 50% or more: use it
```

Nesting if statements is a great way to simplify your code when you have multiple conditions that depend on one. The nested conditions are only evaluated if the parent condition is met. Here's an example in JavaScript:

```
let laptopWorks = true
let batteryLow = true
let internetUnavailable = true
let batteryIsCharged = false

if (laptopWorks) {
  if (batteryLow) {
    console.log('Charge it')
  } else if (internetUnavailable) {
    console.log('Restart the router')
  } else if (batteryIsCharged) {
    console.log('Use it')
  }
}

// output: Charge it
```

Since all the conditions depend on the laptop working, we only need to write it once. If the laptop works, the nested conditions are evaluated. However, if the laptop doesn't work, none of the nested conditions are evaluated. Let's change the value of `laptopWorks` to `false` and run the code again:

```
laptopWorks = false

if (laptopWorks) {
  if (batteryLow) {
    console.log('Charge it')
  } else if (internetUnavailable) {
    console.log('Restart the router')
  } else if (batteryIsCharged) {
    console.log('Use it')
  }
}

// output: (nothing)
```

When nesting, you should always indent the nested conditions to make your code easier to read. If you want to handle the case when the laptop doesn't work, you can add an `else` clause to the parent condition:

```
if (laptopWorks) {
  if (batteryLow) {
    console.log('Charge it')
  } else if (internetUnavailable) {
    console.log('Restart the router')
  } else if (batteryIsCharged) {
    console.log('Use it')
  }
} else {
  console.log('The laptop is broken. It needs fixing')
}

// output: The laptop is broken. It needs fixing
```

When writing conditional statements, you can omit the curly braces if you only have one statement inside the condition:

```
if (laptopWorks) {
  if (batteryLow) console.log('Charge it')
  else if (internetUnavailable)
    console.log('Restart the router')
  else if (batteryIsCharged) console.log('Use it')
} else {
  console.log('The laptop is broken. It needs fixing')
}
```

We could also write the `else` clause without curly braces, but using curly braces makes it easier to read in this case. It's important to prioritize readability over brevity when writing code.

The Switch statement

The switch statement is a way to simplify the process of writing multiple if-else statements. It's a type of conditional statement that allows you to compare a value against multiple cases. The switch statement compares an expression to a list of possible values, and if the expression matches one of the values, the code inside its corresponding case will be executed. Here's what the syntax looks like:

```
switch (expression) {
  case value1:
    // code to be executed if expression matches value1
    break
  case value2:
    // code to be executed if expression matches value2
    break
  default:
  // code to be executed if expression does not match
  // any of the cases
}
```

In the above syntax,

- The `expression` is the value that will be compared against the cases.
- `value1` and `value2` are the possible values the expression can match.
- The `break` keyword is used to stop the execution of the switch statement after the code inside a case is executed. If there's no `break` keyword in a case, the switch statement will continue to run until it reaches a `break` keyword or until it reaches the end.
- The `default` case is executed if the expression does not match any case.

To illustrate how the switch statement works, let's look at an example. Let's say we have a variable called `day` that stores the day of the week, and we want to log different messages to the console depending on the day of the week. We could use an if-else statement to do this:

```
const day = 'Monday'

if (day === 'Monday') {
  console.log('Today is Monday!')
} else if (day === 'Tuesday') {
  console.log('Today is Tuesday!')
} else if (day === 'Wednesday') {
  console.log('Today is Wednesday!')
} else if (day === 'Thursday') {
  console.log('Today is Thursday!')
} else if (day === 'Friday') {
  console.log('Today is Friday!')
} else if (day === 'Saturday') {
  console.log('Today is Saturday!')
} else if (day === 'Sunday') {
  console.log('Today is Sunday!')
} else {
  console.log('That is not a day of the week!')
}

// output: Today is Monday!
```

This code works, but we can simplify it by using a switch statement:

```
const day = 'Monday'

switch (day) {
  case 'Monday':
    console.log('Today is Monday!')
    break
  case 'Tuesday':
    console.log('Today is Tuesday!')
    break
  case 'Wednesday':
    console.log('Today is Wednesday!')
    break
  case 'Thursday':
    console.log('Today is Thursday!')
    break
  case 'Friday':
    console.log('Today is Friday!')
    break
  case 'Saturday':
    console.log('Today is Saturday!')
    break
  case 'Sunday':
    console.log('Today is Sunday!')
    break
  default:
    console.log('That is not a day of the week!')
}

// output: Today is Monday!
```

nested switch statements

When checking for multiple cases of a value or variable, the switch statement is usually more efficient than the if-else statement because it requires fewer lines of code and is easier to read. How-

ever, this is not always the case. If you're working with multiple conditions, the switch statement might become difficult to manage.

For example, if you want to check if the day is a weekday in January using two variables, `day` and `month`, you can write a nested switch statement like this:

```
const day = 'Monday'
const month = 'January'

switch (day) {
  case 'Monday':
    switch (month) {
      case 'January':
        console.log('Today is a weekday in January!')
        break
      default:
        console.log('Today is a weekday!')
    }
    break
  default:
    console.log('That is not a day of the week!')
}

// output: Today is a weekday in January!
```

In the above example:

The first case checks if the value of `day` is 'Monday'; if it is, it enters the inner switch statement, which checks the value of the `month` variable. The inner switch statement checks if the value of `month` is 'January', and if it is, it logs the string 'Today is a weekday in January!' to the console and uses the break statement to exit the switch statement. If the value of `month` is not 'January', it logs the string 'Today is a weekday!' to the console.

While this code works, it's not readable. The nested switch statement is difficult to understand, and it's easy to make a mistake when writing it. This is why it's better to use an if-else statement in this case:

```
const day = 'Monday'
const month = 'January'

if (day === 'Monday' && month === 'January') {
  console.log('Today is a weekday in January!')
} else {
  console.log('That is not a day of the week!')
}

// output: Today is a weekday in January!
```

When you need to nest a switch statement, it's good to consider using an if-else statement instead. Writing complex switch statements can easily lead to mistakes in your code.

It is worth noting that the switch statement is not limited to checking for equality. You can use it to check for other conditions as well. For example, you can use the switch statement to check if a number is greater than or less than a certain value:

```
const num = 5

switch (num) {
  case num > 5:
    console.log('num is greater than 5')
    break
  case num < 5:
    console.log('num is less than 5')
    break
  default:
    console.log('num is equal to 5')
```

```
}

// output: num is equal to 5
```

This method is less common but can be useful in some cases. If you find the switch statement confusing, it's okay to use an if-else statement instead.

Remember always to include a `break` keyword after each case. Otherwise, your switch statement will continue to execute the code blocks for the remaining cases, even after the first match is found:

```
const day = 'Monday'

switch (day) {
  case 'Monday':
    console.log('Today is Monday!')
  case 'Tuesday':
    console.log('Today is Tuesday!')
  case 'Wednesday':
    console.log('Today is Wednesday!')
  case 'Thursday':
    console.log('Today is Thursday!')
  case 'Friday':
    console.log('Today is Friday!')
  case 'Saturday':
    console.log('Today is Saturday!')
  case 'Sunday':
    console.log('Today is Sunday!')
  default:
    console.log('That is not a day of the week!')
}

// output: Today is Monday!
// output: Today is Tuesday!
// output: Today is Wednesday!
```

```
// output: Today is Thursday!
// output: Today is Friday!
// output: Today is Saturday!
// output: Today is Sunday!
// output: That is not a day of the week!
```

The switch statement is usually more structured and easy to read when the cases are simple and well-defined. However, whether a switch statement or an if-else statement is more efficient can depend on the specific use case.

The ternary operator

The ternary operator is a shorthand way to write an if-else statement. It's a conditional operator that takes three operands: a condition, an expression to execute if the condition is true, and an expression to execute if the condition is false. Here's what the syntax looks like:

```
condition ? expression1 : expression2
```

To use a ternary operator, you start by writing a condition followed by a question mark (?). Then, you write an expression that will be executed if the condition is true. Finally, you write a colon (:) and an expression that will be executed if the condition is false. The colon represents the else clause and is required in the ternary operator.

Let's look at an example using a variable that stores a person's age. We want to log a string to the console based on the person's age. If the person is 18 or older, we'll log "You can vote"; if the person is less than 18, we'll log "You can't vote". First, we'll do it using an if-else statement:

```
let age = 18

if (age >= 18) {
  console.log('You can vote')
} else {
  console.log("You can't vote")
}

// output: You can vote
```

Here's how we can do it using the ternary operator:

```
let age = 18

age >= 18
  ? console.log('You can vote')
  : console.log("You can't vote")

// output: You can vote
```

The ternary operator is much shorter than the if-else statement but can be difficult to read in cases like this. It's mainly used when assigning a value to a variable based on a condition:

```
let age = 18
let canVote = age >= 18 ? true : false

console.log(canVote)
// output: true
```

See that the ternary operator is much easier to read here. It's a great way to simplify your code when working with simple conditions.

You can also nest ternary operators:

```
let score = 80

let grade = score >= 90 ? 'A' : score >= 70 ? 'B' : 'C'

console.log(grade)
// output: B
```

In the above example, the ternary operator checks if the value of `score` is greater than or equal to 90. It assigns the string 'A' to the `grade` variable if it is. If it's not, it checks if the value of `score` is greater than or equal to 70. If it is, it assigns the string 'B' to the `grade` variable, else it assigns the string 'C' to the `grade` variable.

Nesting ternary operators can make your code difficult to read, so it's best to avoid doing so when possible. If you find yourself nesting ternary operators, you should consider using an if-else statement instead:

```
let score = 80
let grade = ''

if (score >= 90) grade = 'A'
else if (score >= 70) grade = 'B'
else grade = 'C'
```

Conditional statements are an important part of any programming language. Without them, you wouldn't be able to build programs that can make decisions and perform different actions based on conditions or user input.

Loops

We've seen how to use Array methods like `forEach` and `map` to iterate over the elements in an array. In programming, we call this looping. To loop means to repeat a process until a condition is met. It could be until a specified number of times, a specific action is completed, or a value is reached. It is a fundamental concept in programming and is used in several ways. JavaScript has different types of loops, apart from the array methods we've seen, which allow for more control over the looping process. These are the `for` loop, the `while` loop, and the `do...while` loop. Let's take a look at each of these.

The `for` Loop

Let's say we have an array of numbers, `[1, 2, 3, 4, 5]`, and want to multiply each element by `2`. We could use the `map` method to do this:

```
let numbers = [1, 2, 3, 4, 5]

let doubled = numbers.map((number) => number * 2)

console.log(doubled) // [2, 4, 6, 8, 10]
```

Each time we use the `map` method, the process is the same: we call the method on an array, pass in a callback function, and return a new array.

On the other hand, a for loop allows for more control over the iteration process. With the `for` loop, we can add conditions and

directly manipulate elements in an array rather than returning a new array. Here's how we could use a `for` loop to multiply each element in the numbers array by 2:

```
let numbers = [1, 2, 3, 4, 5]

for (let i = 0; i < numbers.length; i++) {
  numbers[i] = numbers[i] * 2
}

console.log(numbers) // [2, 4, 6, 8, 10]
```

The `for` loop is made up of three parts: the initialization, the condition, and the final expression.

The **initialization** is where we declare a variable and set it equal to a starting value. In this case, we set `i` equal to `0`. We use this variable to keep track of the index of the array, and we can name it whatever we want. `i` is a common variable name for this purpose. The variable is declared with the `let` keyword because we want to be able to change the value of the variable as the loop runs.

The **condition** is a boolean expression that determines the number of times a loop will run and when it will stop. Without it, the loop will continue to execute indefinitely. In this case, the loop will continue to run as long as `i` is less than the length of the numbers array.

The **final expression** is where we increment or decrement the variable. In our example, we increment `i` by 1 in each loop iteration. So the first time the loop runs, `i` is 0. The second time, `i` is 1, and so on. This is how we can access each element in the array. The loop will continue to run until the condition is no longer met.

Since the `for` loop gives us more control over the iteration process, we can also keep the values of the original array and create a new one with the doubled values if we want to. Here's how we could do that:

```
let numbers = [1, 2, 3, 4, 5]

let doubled = []

for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2)
}

console.log(doubled) // [2, 4, 6, 8, 10]
```

Here, we first create an empty array called `doubled`, then we use the `for` loop to iterate over the numbers array. Inside the loop, we push the doubled value of each element into the `doubled` array. In most cases, when iterating over an array, it's easier to use the `map` method. However, the `for` loop may be more appropriate if we need more control over how the array is manipulated.

the `continue` keyword

The `continue` keyword is used to skip the current iteration of a loop and continue with the next one. For example, if we want to print all the numbers from 1 to 10 except for 5. We can use the `continue` keyword to skip the number 5:

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    continue
  }
  console.log(i)
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

The `if` statement checks if the current value of `i` is equal to 5. If it is, the `continue` statement is executed, which skips the rest of the code in the loop and continues with the next iteration.

the `break` keyword

The `break` keyword is used to exit a loop. So if instead, we want to stop the loop when `i` is equal to 5, we can use the `break` keyword:

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    break
  }
  console.log(i)
}
```

Output:

```
1
2
3
4
```

Since the `break` statement ends the loop when `i` is equal to 5, the console.log() statement is not executed for that iteration. So we only get the numbers from 1 to 4 logged to the console.

The `for...in` loop

The `for...in` loop is used to iterate over the enumerable properties of an object. If we have an object with a list of properties and we want to access each property, we could use a `for` loop to iterate over the object's properties like this:

```
let person = {
  name: 'Jane',
  age: 27,
  city: 'Kigali',
}

for (let i = 0; i < Object.keys(person).length; i++) {
  console.log(
    `${Object.keys(person)[i]}: ${
      person[Object.keys(person)[i]]
    }`
  )
}
```

While this works, it's not easy to read. We can use the `for...in` loop to write the same code in a more concise way. Wit the `for...in` loop, we can directly access the keys of an object without having to use the `Object.keys()` method:

```
let person = {
  name: 'Jane',
  age: 27,
  city: 'Kigali',
}

for (let key in person) {
  console.log(`${key}: ${person[key]}`)
}
```

Output:

```
name: Jane
age: 27
city: Kigali
```

In the example above, we declared a variable named `key` and assigned it the value of each property in the `person` object using the `in` operator. We then used the `key` variable to access the value of each property in the `person` object. **The `in` operator in JavaScript is used to check whether a specified property is in an object**. We can also use it outside of a loop:

```
let person = {
  name: 'Jane',
  age: 27,
  city: 'Kigali',
}

if ('name' in person) {
  console.log('The person object has a name property.')
}

// Output: The person object has a name property.
```

We can also use it to check if an element exists in an array:

```
let fruits = ['apple', 'banana', 'orange']

if ('apple' in fruits) {
  console.log(
    'The apple element exists in the fruits array.'
  )
}

// Output: The apple element exists in the fruits array.
```

Since we can use the `in` operator on an array, we can also use the `for...in` loop to iterate over the elements of an array:

```
let names = ['Jane', 'John', 'Ada']

for (let index in names) {
  console.log(names[index])
}
```

Output:

```
Jane
John
Ada
```

When using the `for...in` loop to iterate over an array, the variable assigned to the `in` operator will be the index of each element in the array. An easier way to iterate over the elements of an array is to use the `for...of` loop.

The `for...of` loop

The `for...of` loop is similar to the `for...in` loop, but it iterates over the values of an array instead of the indexes. Let's rewrite the `names` loop example using the `for...of` loop:

```
let names = ['Jane', 'John', 'Ada']

for (let name of names) {
  console.log(name)
}
```

Output:

```
Jane
John
Ada
```

When accessing the elements of an array using the `for...of` loop, the variable assigned to the `of` operator will be the value of each element in the array. We can also use the `for...of` loop to iterate over the characters of a string:

```
let name = 'Jane'

for (let character of name) {
  console.log(character)
}
```

Output:

```
J
a
n
e
```

When working with arrays, the `for...of` loop is easier to read and understand than the `for...in` loop. However, we can't use the `for...of` loop to directly access the properties of an object as we did with the `for...in` loop.

The `while` Loop

The `while` loop is similar to the `for` loop but doesn't have a final expression. It's used when we don't know how many times a loop will run. For example, if we have a dice game where we roll a die until we get a 6. We could get a 6 on the first roll, or it could take 10 rolls. Since we don't know how many times, we can't use a `for` loop. With the `while` loop, you can keep rolling the die as long as the value is not 6. In plain English, writing the `while` loop would look like this:

```
While the value of the die is not 6, roll the die.
```

Let's see how we could write the `while` loop in JavaScript. We'll use the `Math.random()` method to generate a random number from 1 to 6, then the `Math.ceil()` method to round the number up to the nearest integer. First, we'll create a variable named `dieRoll`, then we'll use the `while` loop to update its value until our `Math` methods return a 6:

```
let dieRoll = 0

while (dieRoll !== 6) {
  dieRoll = Math.ceil(Math.random() * 6)
  console.log(`You rolled a ${dieRoll}`)
}
```

Depending on how many rolls it takes to get a 6, the code should return something like this (but not exactly since the numbers are random):

```
You rolled a 2
You rolled a 4
You rolled a 3
You rolled a 6
```

If the first roll is a 6, the loop will run only once. If it takes 10 rolls, the loop will run 10 times. The `while` loop will continue to run as long as the condition is true. In this case, the condition is that `dieRoll` is not equal to 6. So once the value of `dieRoll` is 6, the loop will stop running. In a `while` loop, the condition is checked before the loop runs. If we had assigned `dieRoll` a value of 6 when we declared it, the loop would never run because the condition would be false from the start:

```
let dieRoll = 6

while (dieRoll !== 6) {
  dieRoll = Math.ceil(Math.random() * 6)
  console.log(`You rolled a ${dieRoll}`)
}
```

In a `while` loop, we must ensure that the condition will eventually be met. Otherwise, the loop will run indefinitely. See how we updated the value of `dieRoll` in every iteration. If we didn't do that, `dieRoll` would always be 0 in our first example, and the condition `dieRoll !== 6` would always be true, which would cause the loop to run forever. When a loop runs indefinitely, it's called an **infinite loop**.

Let's try removing the line that updates the value of `dieRoll`. Running an infinite loop like this will cause your browser to freeze or crash, so keep your cursor close to the `X` button (for closing the tab) so that you can stop it from crashing your browser. Don't let it run for too long, or it'll become unresponsive:

```
let dieRoll = 0

while (dieRoll !== 6) {
  console.log(`You rolled a ${dieRoll}`)
}
```

Congrats on running your first infinite loop! You'll come across several infinite loops in your JavaScript journey. When you do, it means that there's a condition that's always true.

The `do...while` Loop

A `do...while` loop is similar to a `while` loop, but unlike the `while` loop, the condition is checked after the loop runs. This means that the loop will always run at least once. So in our second example with the dice game, where we assigned `dieRoll` a value of 6 upon declaration, the `do...while` loop would still run once:

```
let dieRoll = 6

do {
  dieRoll = Math.ceil(Math.random() * 6)
  console.log(`You rolled a ${dieRoll}`)
} while (dieRoll !== 6)
```

number-guessing-game

The `do...while` loop is useful when we want to run a loop at least once, regardless of the condition. Let's create a number-guessing game where a user has to guess a number between 1 and 10. We'll use the Math methods to generate a random number between 1 and 10, then we'll ask the user to guess the number. If they guess correctly, the game ends, and we'll congratulate them. If they guess

incorrectly, we'll ask them to guess again. We can get a user's input with the `prompt()` method, a built-in browser function that displays a dialog box with a text input field. We'll use the `do...while` loop to keep asking the user to guess the number until they guess correctly:

```
let correctNumber = Math.ceil(Math.random() * 10)
let guess = 0

do {
  guess = prompt('Guess a number between 1 and 10')
  if (guess == correctNumber) {
    alert('You guessed the correct number!')
  } else {
    alert('Sorry, that was incorrect. Try again.')
  }
} while (guess != correctNumber)
```

In this example, we first generate a random number between 1 and 10 and assign it to the `correctNumber` variable. Then we declare a variable for storing the user's guess and assign it a value of 0. We use the `do...while` loop to keep asking the user to guess the number until they guess correctly.

Inside the loop, we use the `prompt()` method to ask the user to guess the number. Assigning the `prompt()` method to the `guess` variable stores the user's input in it.

We then check if the user's guess is equal to the correct number. If it is, we congratulate them and end the loop. If it's not, we tell them that they guessed incorrectly and ask them to guess again. The `alert()` method is another built-in browser function that displays a simple dialog box with a message. The `do...while` loop ensures that the code inside the loop is executed at least once before the condition is checked. So even if we use the `correctNumber` variable as the value of `guess`, the user will still be asked to guess the number again:

```
let correctNumber = Math.ceil(Math.random() * 10)
let guess = correctNumber

do {
  guess = prompt('Guess a number between 1 and 10')
  if (guess == correctNumber) {
    alert('You guessed the correct number!')
  } else {
    alert('Sorry, that was incorrect. Try again.')
  }
} while (guess != correctNumber)
```

On the other hand, if we use a `while` loop, the user will never be asked to guess the number because the condition is false from the start:

```
let correctNumber = Math.ceil(Math.random() * 10)
let guess = correctNumber

while (guess != correctNumber) {
  guess = prompt('Guess a number between 1 and 10')
  if (guess == correctNumber) {
    alert('You guessed the correct number!')
  } else {
    alert('Sorry, that was incorrect. Try again.')
  }
}
```

We could fix this by initializing the `guess` variable with a value of `0`. However, if the expression in the `correctNumber` variable declaration returns `0` in the first place, the `while` loop will still not run.

Notice that we used the regular equality operator (`==`) to check if the user's guess is equal to the correct number and not the strict equality operator (`===`). This is because the `prompt()` method

returns a string, and the `correctNumber` variable is a number. Since the strict equality operator checks if both the type and the value are the same, the condition would always be false if we used it.

It's worth noting that the `alert()` and `prompt()` methods are considered blocking methods, which means that the code execution will stop until the dialog box is closed. We also won't be able to interact with the web page or use the browser console while it is open. We'll talk about other non-blocking ways we can get user input and display messages in later chapters.

Try to rewrite each example on your own to gain a deeper understanding of how loops work before moving on to the next section. Additionally, you can challenge yourself by creating another interactive game using loops and the `prompt()` and `alert()` methods. This will not only help you become more comfortable with loops, but it will also enhance your JavaScript skills.

The Document Object Model (DOM)

The Document Object Model (DOM) is an object-oriented representation of a web page, where each element of the page, such as a button, image or text, is represented as an object with its own set of properties and methods. The DOM is a programming interface for HTML and XML documents, that allows us to use JavaScript to access and update the content, structure, and style of a web page. With the DOM, we can listen to events, like a click on a button and respond to them. We can also create new elements and attributes, change the content of existing elements, or remove elements from the page.

The DOM represents a document as a tree of nodes. The word "node" refers to a single point in a data structure, and in the case of the DOM, it represents an element or a piece of text on the web page. Each node is an object representing a part of the document. The tree-like structure of the DOM is formed by parent-child relationships between the nodes, with each element containing other elements, which can be thought of as child nodes.

For example, in an HTML document, a node would represent an element such as a paragraph `<p>Hello World</p>`, and the text "Hello World" would be a child node of the paragraph node. Nodes provide a way to represent the web page in a logical and hierarchical way (from the top of the page to the bottom), making it easy to navigate and understand the structure of the document.

Let's take a look at a simple HTML document:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Simple Web Page</title>
  </head>
  <body>
    <h1>This is awesome!</h1>
  </body>
</html>
```

The DOM tree for this document would look like this:

```
                Document
                 /    \
              html    doctype
              /   \
          head      body
            |         |
          title       h1
            |         |
          textNode  textNode
```

In this tree representation, the `Document` is the root node, and it has two children: the `html` node and the `doctype` node. The `html` node is the container for all the other elements on the web page, and it has two children: the `head` and the `body`. The `head` node contains the `title` element, which in turn has a text node child with the text "My Simple Web Page". The `body` node has one child: the `h1` element. The `h1` element contains a text node child with the text "This is awesome!"

Accessing the DOM

Using JavaScript to access the DOM is straightforward. First, let's create an HTML document. Paste the following code into a file called `index.html`, then open it in your browser:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learning the DOM</title>
  </head>
  <body>
    <h1>Using JavaScript to access the DOM</h1>
    <p>DOM stands for Document Object Model</p>
  </body>
</html>
```

Now, open the browser's developer tools and navigate to the console. This is where we will write our JavaScript code.

Through the DOM, we can access the elements of our webpage, including the `title`, `h1` and `p` elements. First, we need to get a reference to the element we want to access. The browser provides a global object called `document` that represents the DOM, which we can use to access all elements of our web page. For example, to access the `title` element, we can use the `document` object's `title` property. In the browser console of your web page, type the following code and hit Enter:

```
document.title
```

This will return the text "Learning the DOM". The `title` property of the `document` object is a reference to the `title` element of the web page. We can also access the `head` and `body` elements by using the `document` object's `head` and `body` properties:

```
document.head
// Output: <head>...</head>

document.body
// Output: <body>...</body>
```

The `getElementsByTagName` method

The `document` object has a `getElementsByTagName` method that can be used to access elements by their tag name. This method takes a string argument, which is the name of the tag. To access the `h1` element, type the following code in the console:

```
document.getElementsByTagName('h1')
```

This will return an HTMLCollection containing a reference to the `h1` element. An HTMLCollection is an array-like object that contains references to the elements of a web page. When we use the `getElementsByTagName` method, it returns an HTMLCollection containing references to all elements with the specified tag name in a hierarchical order. Since we have only one `h1` element in our document, the HTMLCollection contains only one element. Here's what the console output looks like:

```
HTMLCollection[h1]
```

If we had two `h1` elements, the HTMLCollection would contain two elements:

```
HTMLCollection[(h1, h1)]
```

To access the `h1` element in the HTMLCollection, we can use the bracket notation, just like we would with an array:

```
document.getElementsByTagName('h1')[0]
```

This will return the `h1` element:

```
<h1>Using JavaScript to access the DOM</h1>
```

We can also access the `h1` element by using the `item` method of the HTMLCollection, which takes an index as an argument:

```
document.getElementsByTagName('h1').item(0)

// Output: <h1>Using JavaScript to access the DOM</h1>
```

Now that we have access to the `h1` element, we can get the text content of the `h1` element by using the `textContent` or `innerText` property:

```
document.getElementsByTagName('h1')[0].textContent
// Output: "Using JavaScript to access the DOM"

document.getElementsByTagName('h1')[0].innerText
// Output: "Using JavaScript to access the DOM"
```

The `textContent` and `innerText` properties are used to get or set the text content of an HTML element. The `innerText` property takes into account the styles applied to the text, such as visibility, and will not return the text if it is hidden by CSS, while the `textContent` property will return the text regardless of the styles applied to it. Since the DOM properties are browser-dependent, some browsers may return different results in certain cases, or have different levels of support for the properties. But in general, the behavior of these properties is similar across browsers.

Let's use the `textContent` property to change the text of the `h1` element. In your console, type the following code:

```
document.getElementsByTagName('h1')[0].textContent =
  'DOM Manipulation'

// Output: DOM Manipulation
```

This will change the text of the `h1` element to "DOM Manipulation". We can also assign the `h1` element to a variable so that we don't have to keep typing `document.getElementsByTagName('h1')[0]` every time we want to access it:

```
const pageHeading =
  document.getElementsByTagName('h1')[0]

console.log(pageHeading.textContent)
// Output: DOM Manipulation
```

Now, we can use the `pageHeading` variable to access the `h1` element, and even change its text. Let's use the `innerText` property this time:

```
pageHeading.innerText =
  'Learning the DOM with JavaScript'

// Output: Learning the DOM with JavaScript
```

Running this will make the text of the `h1` element "Learning the DOM with JavaScript".

We can access any other tag in the same way. Here's another example with the `p` element:

```
document.getElementsByTagName('p')[0].textContent

// Output: DOM stands for Document Object Model
```

Accessing elements more specifically with `getElementById` and `getElementsByClassName`

So far, you've seen how to access elements with a certain tag by using the `getElementsByTagName` method. However, most of the time, you'll need to access elements more specifically. For example, if you have multiple `p` elements that represent different content on your web page, it'll be difficult to use the `getElementsByTagName` method to identify the `p` element you want, since it returns an HTMLCollection of `p` tags without any direct information about their content. Instead, you can access elements more specifically by using the `id` or `class` attributes.

The `getElementById` method

The `getElementById` method of the `document` object is used to access an element by its `id` attribute. Since the HTML `id` attribute is unique, we can use it to directly identify a specific element on a web page. Let's add another `p` element to our web page, and give both `p` elements an `id` attribute. Modify your `index.html` file to look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learning the DOM</title>
  </head>
  <body>
    <h1>Using JavaScript to access the DOM</h1>
    <p id="dom-definition">
      DOM stands for Document Object Model
    </p>
    <p id="dom-example">
      We can use the DOM to manipulate the content of a
      web page
```

```
    </p>
  </body>
</html>
```

Now, we can access the `p` elements by using the `getElementById` method of the `document` object. This method takes a string argument, which is the value of the `id` attribute of the element we want to access. To access the `p` element with the `id` attribute of "dom-definition", type the following code in your browser console:

```
document.getElementById('dom-definition')

// Output: <p id="dom-definition">DOM stands for Document
// Object Model</p>
```

See that this directly returns the `p` element with the `id` attribute of "dom-definition". We can also access its text content by using the `textContent` property:

```
document.getElementById('dom-definition').textContent

// Output: DOM stands for Document Object Model
```

Let's also get the text content of the `p` element with `id` attribute of "dom-example":

```
document.getElementById('dom-example').textContent

// Output: We can use the DOM to manipulate the content
// of a web page
```

With the `id` attribute and the `getElementById` method, it's easy to access a specific element on a web page. However, if you have multiple elements with the same `id` attribute, the `getElementById`

method will only return the first element it finds. To identify multiple elements in the same category, you can use the `class` attribute instead, together with the `getElementsByClassName` method.

The `getElementsByClassName` method

The `getElementsByClassName` method of the `document` object is used to access elements by their `class` attribute. The `class` attribute is not unique, so this method returns an HTMLCollection of elements with the same `class` attribute. Let's give the `p` elements in our `index.html` file a `class` attribute of "paragraph":

```
<p id="dom-definition" class="paragraph">
  DOM stands for Document Object Model
</p>
<p id="dom-example" class="paragraph">
  We can use the DOM to manipulate the content of a web
  page
</p>
```

Now, we can access all elements with the `class` attribute of "paragraph" by using the `getElementsByClassName` method. This method takes a string argument, which is the value of the `class` attribute of the elements we want to access. Run the following code in your browser console:

```
document.getElementsByClassName('paragraph')

// Output: HTMLCollection(2) [p#dom-definition.paragraph,
// p#dom-example.paragraph]
```

In the output `p#dom-definition.paragraph`, the `p` represents the tag name, `#` represents the `id` attribute, and `.` represents the `class` attribute. So `#dom-definition` means that the element has an `id` attribute of "dom-definition", and `.paragraph` means that the element

has a `class` attribute of "paragraph". The `getElementsByClassName` method returns an HTMLCollection of all elements with the specified `class` attribute; in this case, 'paragraph'.

Accessing elements with the `querySelector` and `querySelectorAll` methods

The `querySelector` and `querySelectorAll` methods are similar to the `getElementById` and `getElementsByClassName` methods, but they allow us to access elements more flexibly. With these methods, we can access elements using either their `id`, `class` or any other attribute. We can also access elements by their tag name, or by a combination of their tag name and attribute. Let's see how these methods work.

The `querySelector` method of the `document` object is used to access the first element that matches a specified selector. A selector is a string that specifies the element or elements to be selected. For example, the selector `p` will select all `p` elements on a web page. The `querySelector` method takes the selector as an argument and returns the first element that matches it. Let's use the `querySelector` method to select the first `p` element on our web page:

```
1  document.querySelector('p')
2
3  // Output: <p id="dom-definition" class="paragraph">
4  // DOM stands for Document Object Model</p>
```

On the other hand, the `querySelectorAll` method returns a **NodeList** of all elements that match the specified selector. A NodeList is similar to an HTMLCollection, but is a more general list that can contain elements of different node types including text nodes, comment nodes, and document nodes. Both a NodeList and an HTMLCollection are array-like objects, so we can access their elements by index.

Let's use the `querySelectorAll` method to select all `p` elements on our web page:

```
document.querySelectorAll('p')

// Output: NodeList(2) [p#dom-definition.paragraph,
// p#dom-example.paragraph]
```

We can also use the `querySelector` and `querySelectorAll` methods to select elements by their `id` or `class` attributes. For example, the selector `#dom-definition` will select the element with the `id` attribute of "dom-definition", and the selector `.paragraph` will select all elements with the `class` attribute of "paragraph". Let's use the `querySelector` method to select the element with the `id` attribute of "dom-definition":

```
document.querySelector('#dom-definition')

// Output: <p id="dom-definition" class="paragraph">
// DOM stands for Document Object Model</p>
```

And let's use the `querySelectorAll` method to select all elements with the `class` attribute of "paragraph":

```
document.querySelectorAll('.paragraph')

// Output: NodeList(2) [p#dom-definition.paragraph,
// p#dom-example.paragraph]
```

To access elements even more specifically, we can use a combination of their tag name and attribute. For example, the selector `p#dom-definition` will select the `p` element with the `id` attribute of "dom-definition" and the selector `p.paragraph` will select all `p` elements with the `class` attribute of "paragraph". So if another

element with the same class or id exists, the `querySelector` method will only return the element that matches both the tag name and the attribute. Let's add a `div` element with the same `class` attribute of "paragraph" as the `p` elements to our `index.html` file:

```
<body>
  <h1>Using JavaScript to access the DOM</h1>
  <p id="dom-definition" class="paragraph">
    DOM stands for Document Object Model
  </p>
  <p id="dom-example" class="paragraph">
    We can use the DOM to manipulate the content of a web
    page
  </p>
  <div class="paragraph">
    This div element has the same class attribute as the
    p elements
  </div>
</body>
```

Now, let's use the `querySelector` method to select the `div` element with the `class` attribute of "paragraph". Run the following code on your browser console:

```
document.querySelector('div.paragraph')

// Output: <div class="paragraph">This div element has
// the same class attribute as the p elements</div>
```

The `querySelector` method returns only the element that matches both the tag name and the attribute. Let's get all `p` elements with the `class` attribute of "paragraph" using the `querySelectorAll` method:

```
document.querySelectorAll('p.paragraph')

// Output: NodeList(2) [p#dom-definition.paragraph,
// p#dom-example.paragraph]
```

Here's an example with the `id` attribute. We'll use the selector `p#dom-definition` to get the `p` element with `id` of "dom-definition":

```
document.querySelector('p#dom-definition')

// Output: <p id="dom-definition" class="paragraph">
// DOM stands for Document Object Model</p>
```

When using the `querySelector` and `querySelectorAll` methods to access elements by their `id` or `class` attributes, you must include the `#` or `.` symbol before the attribute value. `#` represents the `id` attribute and `.` represents the `class` attribute. If you don't include them before the attribute value, the methods will assume that you're trying to access an element by its tag name. So if instead of using "#dom-definition" we used "dom-definition", the `querySelector` method would search for an element with the tag name of "dom-definition" and return `null`:

```
document.querySelector('dom-definition')

// Output: null
```

Accessing element attributes and style properties

The `attributes` property and the `getAttribute` method‘

We can access the attributes of an element using the `getAttribute` method and the `attributes` property. The `getAttribute` method

takes the name of the attribute as an argument and returns the value of the attribute. The `attributes` property returns an array-like object of all the attributes of an element. Let's use the `getAttribute` method to get the value of the `id` attribute of the `p` element with the `id` of "dom-definition":

```
document
  .querySelector('#dom-definition')
  .getAttribute('id')

// Output: "dom-definition"
```

And let's use the `attributes` property to get all the attributes of the `p` element with the `id` of "dom-definition":

```
document.querySelector('#dom-definition').attributes

// Output: NamedNodeMap(2) [id, class]
```

The `attributes` property returns a **NamedNodeMap** object that contains all the attributes of an element. *A NamedNodeMap represents a collection of nodes that can be accessed by name.* We can access the attributes of a NamedNodeMap object using the square bracket notation and the index of the attribute or by using the dot notation and the name of the attribute. Let's use the square bracket notation to get the `id` attribute of the `p` element with the `id` of "dom-definition":

```
document.querySelector('#dom-definition').attributes[0]

// Output: id
```

Here's the same example using the dot notation:

```
document.querySelector('#dom-definition').attributes.id

// Output: id
```

To access the value of an attribute, we can use its `value` property:

```
document.querySelector('#dom-definition').attributes[0]
  .value
// Output: "dom-definition"

document.querySelector('#dom-definition').attributes.id
  .value
// Output: "dom-definition"
```

The `style` property

The `style` property returns an object that contains all the style properties of an element. Let's get the `style` property of our `dom-definition` element:

```
document.querySelector('#dom-definition').style

// output: CSSStyleDeclaration {accentColor: '',
// backgroundColor: '', fontSize: '', ...}
```

Notice that the `style` property returns a CSSStyleDeclaration object. A CSSStyleDeclaration object represents a CSS declaration block. In the JavaScript object, each style property is represented in camelCase. So instead of `background-color` like in CSS, the style property is `backgroundColor`. Since we've not styled the elements, each style property has an empty string as its value.

To directly access the value of a style property, we can use the dot or square bracket notation:

```
document.querySelector('#dom-definition').style
  .backgroundColor
// Output: ""

document.querySelector('#dom-definition').style[
  'backgroundColor'
]
// Output: ""
```

We can also use the `getPropertyValue` method to get the value of a style property:

```
document
  .querySelector('#dom-definition')
  .style.getPropertyValue('background-color')

// Output: ""
```

Modifying DOM Elements

Modifying elements in the DOM is the same as modifying elements in a JavaScript object. We can reassign the value of a DOM element's properties, and add or remove properties. For example, we can change the value of an element's `id` attribute by reassigning the value of the `id` property. Let's change the `id` attribute of the `p` element with the `id` of "dom-definition" to "new-dom-definition":

```
document.querySelector('#dom-definition').id =
  'new-dom-definition'

document
  .querySelector('#new-dom-definition')
  .getAttribute('id')
// Output: "new-dom-definition"
```

DOM elements also have a `setAttribute` method that we can use to modify the value of an attribute. The `setAttribute` method takes two arguments: the name of the attribute and the value of the attribute. Let's change the `id` attribute back to "dom-definition" using the `setAttribute` method:

```
document
  .querySelector('#new-dom-definition')
  .setAttribute('id', 'dom-definition')

document
  .querySelector('#dom-definition')
  .getAttribute('id')
// Output: "dom-definition"
```

Element styles can also be modified by reassigning the value of a style property. Let's change the background color of the `dom-definition` element to red:

```
document.querySelector(
  '#dom-definition'
).style.backgroundColor = 'red'

document.querySelector('#dom-definition').style
  .backgroundColor
// Output: "red"
```

This should also change the background color of the element in the browser. Like the `setAttribute` method, the `style` property has a `setProperty` method that we can use to modify the style of an element. Let's modify the font size of the `dom-definition` element to 3em:

```
document
  .querySelector('#dom-definition')
  .style.setProperty('font-size', '3em')

document.querySelector('#dom-definition').style.fontSize
// Output: "3em"
```

You can see that the browser updates the page whenever we modify the DOM. This is because the DOM is a live representation of the page and any changes made to the DOM are immediately reflected in the browser.

The `innerHTML` property

The `innerHTML` property returns the HTML content of an element in string form. For example, running `document.body.innerHTML` on your browser console will return the HTML content of the `body` element:

```
document.body.innerHTML

// Output: '<h1>...</h1> <p>...</p>..."
```

We can also use the `innerHTML` property to modify the HTML content of an element. Let's add a `p` element with the text "The DOM is awesome!" to the `body` element:

```
document.body.innerHTML += '<p>The DOM is awesome!</p>'
```

Notice that we're appending the new HTML content with the `+=` operator instead of reassigning the value of the `innerHTML` property. If we used the `=` operator, the new HTML content would replace the existing HTML content. Remember that our HTML document has a `div` element with the class of "paragraph" and the text "This div element has the same class attribute as the p elements". Let's use the `innerHTML` property to replace its content with a button:

```
document.querySelector('div.paragraph').innerHTML =
  '<button>Click me!</button>'
```

Now you should see a button in the div instead of the text "This div element has the same class attribute as the p elements".

Creating DOM Elements

The document object has a `createElement` method that we can use to create new elements. The method takes one argument, which is the tag name of the element to be created, such as `p` or `div`. Let's create a new `p` element as an example:

```
document.createElement('p')

// Output: <p></p>
```

The `createElement` method returns a new element with the specified tag name but does not add it to the DOM. We can use the `appendChild` method to add elements to the DOM. The `appendChild` method takes an element as its argument and adds it as a child of the element on which it was called. The new element will be added as the last child of the parent element.

Before we add the `p` element to the DOM, let's add some text to it. We can use the `textContent` or `innerText` property to add text to an element. We'll have to assign the new element to a variable so that we can easily update its properties:

```
let newParagraph = document.createElement('p')

newParagraph.textContent =
  'JavaScript is a versatile language.'
```

We can also give the new element any attributes that we want. Let's give it an `id` attribute with the value of "about-javascript":

```
newParagraph.setAttribute('id', 'about-javascript')
```

Now, we can use the `appendChild` method to add the new element to the DOM:

```
document.body.appendChild(newParagraph)
```

You should see the new paragraph added to the bottom of the page.

Other methods for creating and adding new elements to the DOM include:

insertBefore

The `insertBefore` method adds a new element before a specified element. The method takes two arguments: the element to be added and the element before which the new element will be added. For example, let's add a new `h2` element before the `p` element with the `id` of "dom-example":

```
let newHeading = document.createElement('h2')
newHeading.textContent = 'Working with the DOM'

document.body.insertBefore(
  newHeading,
  document.querySelector('#dom-example')
)
```

replaceChild

The `replaceChild` method replaces an element with a new element. The method takes two arguments: the element to be added and the element to be replaced. Let's replace the `#dom-example` element with a new `div` element. We'll also create a new `p` element inside the `div`:

```
let newDiv = document.createElement('div')
let newParagraph = document.createElement('p')

newParagraph.setAttribute('id', 'dom-example')
newParagraph.textContent = 'Working with the DOM is fun!'
newDiv.appendChild(newParagraph)

document.body.replaceChild(
  newDiv,
  document.querySelector('#dom-example')
)
```

This should replace the `#dom-example` element with the new `div` element. Our new `div` also has a `p` element with the text "Working with the DOM is fun!" and an `id` of "dom-example".

insertAdjacentHTML

The `insertAdjacentHTML` method inserts HTML content at a specified position relative to the element on which it is called. The method takes two arguments: the position and the HTML content. The position can be one of the following strings:

- `beforebegin`: Before the element itself.
- `afterbegin`: Just inside the element, before its first child.
- `beforeend`: Just inside the element, after its last child.
- `afterend`: After the element itself.

Let's add a new `p` element with the text "The DOM is awesome!" before the `#dom-example` element:

```
1  document
2    .querySelector('#dom-example')
3    .insertAdjacentHTML(
4      'beforebegin',
5      '<p>The DOM is awesome!</p>'
6    )
```

This should add a new `p` element with the text "The DOM is awesome!" before the `#dom-example` element.

insertAdjacentElement

The `insertAdjacentElement` method is similar to the `insertAdjacentHTML` method. The only difference is that the

`insertAdjacentElement` method takes an element as its second argument instead of HTML content. Let's add a new `p` element with the text "The DOM is really cool!" after the `#dom-example` element:

```
let newParagraph = document.createElement('p')
newParagraph.textContent = 'The DOM is really cool!'
document
  .querySelector('#dom-example')
  .insertAdjacentElement('afterend', newParagraph)
```

This should add a new `p` element with the text "The DOM is really cool!" after the `#dom-example` element.

`insertAdjacentText`

Like the other `insertAdjacent` methods, the `insertAdjacentText` method inserts text at a specified position relative to the element on which it is called. The method takes two arguments: the position and the text to be inserted. Let's a new text node with the text "JavaScript is for everyone!" inside the `body` element but as its last child:

```
document.body.insertAdjacentText(
  'beforeend',
  'JavaScript is for everyone!'
)
```

This should add the new text node as the last child of the `body` element.

`cloneNode`

The `cloneNode` method returns a copy of the element on which it is called. It takes a boolean argument that determines whether the

copy will include the element's children. Let's create a copy of the `#dom-definition` element:

```
document.querySelector('#dom-definition').cloneNode(true)

// Output: <p id="dom-definition">DOM stands for Document
// Object Model</p>
```

The `cloneNode` method creates a duplicate of an element but does not add it to the DOM. If the element has children, the `cloneNode` method will only copy them if the boolean argument is `true`. So in the example above, the text node child of the `#dom-definition` element was also copied. Let's create another clone of the `#dom-definition` element but without its children this time:

```
document
  .querySelector('#dom-definition')
  .cloneNode(false)

// Output: <p id="dom-definition"></p>
```

createTextNode

The `createTextNode` method creates a text node. It takes one argument, which is the text that will be added to the text node. Let's create a text node with the text "Learning the DOM requires practice". We'll append it to the `#dom-example` element:

```
let textNode = document.createTextNode(
  ' Learning the DOM requires practice'
)
document
  .querySelector('#dom-example')
  .appendChild(textNode)
```

You should see the new text appended to the `#dom-example` element. We added a space before the text so that it would be separated from the current text in the `#dom-example` element.

Removing elements from the DOM

We can use the `removeChild` method to remove an element from the DOM. The `removeChild` method takes one argument, which is the element to be removed. Let's remove the `#dom-definition` element:

```
document.body.removeChild(
  document.querySelector('#dom-definition')
)
```

We can also use the `innerHTML` property to set the value of an element to an empty string. This will remove all of the element's children. Let's remove the `#dom-example` element:

```
document.querySelector('#dom-example').innerHTML = ''
```

Remember that the DOM is a live representation of the web page, so any changes you make to it will be reflected in the browser. However, when you refresh the page, the DOM will be reset to its original state.

Traversing the DOM

To traverse the DOM means to move through the elements of the DOM Tree. Let's make some changes to our `index.html` file so that we can have some more elements to work with. Replace the content of your file with the following code, then refresh the page in your browser:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learning the DOM</title>
  </head>
  <body>
    <h1>Using JavaScript to access the DOM</h1>
    <div id="dom-intro">
      <p id="dom-definition">
        DOM stands for Document Object Model
      </p>
      <p id="dom-example">
        We can use the DOM to manipulate the content of a
        web page
      </p>
    </div>
  </body>
</html>
```

In the new HTML code, we wrapped the `p` elements in a `div` element with an `id` of "dom-intro". Let's see how we can access the DOM elements using their relationship to one another.

The `parentNode` property

The `parentNode` property returns the parent node of the element on which it is called. Let's use the `parentNode` property of the

`#dom-definition` element to access its parent node:

```
document.querySelector('#dom-definition').parentNode

// Output: <div id="dom-intro">...</div>
```

We can also chain the `parentNode` property to access an element's grandparent node:

```
document.querySelector('#dom-definition').parentNode
  .parentNode

// Output: <body>...</body>
```

The `childNodes` property

The `childNodes` property returns a `NodeList` of all child nodes of an element, including element nodes and non-element nodes like text nodes.

Here's an example using the `childNodes` property of the `#dom-intro` element to access its child nodes:

```
document.querySelector('#dom-intro').childNodes

// Output: NodeList(5) [text, p#dom-definition, text,
// p#dom-example, text]
```

The child nodes include:

- A text node with the value "\n " (representing the linebreak between the elements on the page.)
- A `<p>` element node with the id of "dom-definition"
- A text node with the value "\n "
- A `<p>` element node with the id of "dom-example"
- A text node with the value "\n "

The `children` property

The `children` property is similar to the `childNodes` property but it returns an `HTMLCollection` of only the child element nodes:

```
document.querySelector('#dom-intro').children

// Output: HTMLCollection(2) [p#dom-definition,
// p#dom-example]
```

The `firstChild` and `lastChild` properties

The `firstChild` property returns the first child node of an element, including non-element nodes. The `lastChild` property returns the last child node of an element, also including non-element nodes. Let's use the `firstChild` and `lastChild` properties of the `#dom-intro` element to access its first and last child nodes:

```
document.querySelector('#dom-intro').firstChild
// Output: #text

document.querySelector('#dom-intro').lastChild
// Output: #text
```

The `firstElementChild` and `lastElementChild` properties

The `firstElementChild` and `lastElementChild` properties return the first and last child element nodes of an element respectively. Unlike the `firstChild` and `lastChild` properties, only element nodes are returned:

```
document.querySelector('#dom-intro').firstElementChild
// Output: <p id="dom-definition">DOM stands for
// Document Object Model</p>

document.querySelector('#dom-intro').lastElementChild
// Output: <p id="dom-example">We can use the DOM to
// manipulate the content of a web page</p>
```

The `nextSibling` and `previousSibling` properties

The `nextSibling` property returns the next sibling node of an element, including non-element nodes. The `previousSibling` property returns the previous sibling node of an element, also including non-element nodes. Let's use the `nextSibling` and `previousSibling` properties of the `#dom-definition` element to access its next and previous sibling nodes:

```
document.querySelector('#dom-definition').nextSibling
// Output: #text

document.querySelector('#dom-definition').previousSibling
// Output: #text
```

The `nextElementSibling` and `previousElementSibling` properties

Unlike the `nextSibling` and `previousSibling` properties, the `nextElementSibling` and `previousElementSibling` properties return only element nodes:

```
document.querySelector('#dom-definition')
  .nextElementSibling
// Output: <p id="dom-example">We can use the DOM to
// manipulate the content of a web page</p>

document.querySelector('#dom-definition')
  .previousElementSibling
// Output: null
```

Conclusion

The Document Object Model is a powerful tool that allows us to make our web pages dynamic and interactive. It's an essential part of web development, and it's important to understand how it works. Create some more HTML files and use them to practice accessing and modifying the DOM with JavaScript. You can also visit your favorite websites and use the browser's developer tools to explore the DOM.

How to include JavaScript in an HTML file

So far, we've practiced JavaScript by running it in the console. But as we begin to work with web pages and the DOM, we'll need to include JavaScript in our HTML files. We can do this by using the `<script>` tag to write our JavaScript code directly in the HTML file, or by creating a separate JavaScript file with the `.js` extension and linking to it in the HTML file. Let's look at both of these options.

Writing JavaScript in the HTML file (inline JavaScript)

The simplest way to include JavaScript in an HTML file is by adding the code directly to the HTML document using the `<script>` tag. This is usually referred to as inline JavaScript. We can include the JavaScript code in the head or body section of the document, but it's usually best to put it in the body section so that the HTML content is loaded before the JavaScript code. Let's look at an example. Create a new HTML file called `js-in-html.html` and add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JS in HTML</title>
  </head>
  <body>
    <h1>JS in HTML</h1>
    <script>
      alert('Hey everyone!')
    </script>
  </body>
</html>
```

In the example above, we've added an `alert()` method call to the HTML file. When the page loads, the browser will run the JavaScript code and display the alert box with the message "Hey everyone!". You can see this in action by opening the HTML file in your browser.

While inline JavaScript can be a quick solution for small projects or adding simple functionality, it's not the best option for larger projects. If we want to include a lot of JavaScript code in our HTML file, it can become difficult to read and maintain. It's also not very efficient, as the browser will have to parse the JavaScript code every time the page loads. For these reasons, it's usually better to create a separate JavaScript file and link to it in the HTML file.

Creating a separate JavaScript file (external JavaScript)

The most common way to include JavaScript in an HTML file is by creating a separate JavaScript file with the `.js` extension and linking to it in the HTML file. Let's look at an example. In the

same folder as the `js-in-html.html` file, create a new JavaScript file called `js-in-html.js` and add the alert method call to it:

```
alert('Hey everyone!')
```

Now, open the `js-in-html.html` file, remove the existing `<script>` element and add a new one that links to the `js-in-html.js` file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JS in HTML</title>
  </head>
  <body>
    <h1>JS in HTML</h1>
    <script src="./js-in-html.js"></script>
  </body>
</html>
```

In our new example, we've added a `<script>` element that links to the `js-in-html.js` file using its `src` attribute and path, instead of the initial `<script>` element that contained the JavaScript code. When the page loads, the browser will run the JavaScript code in the `js-in-html.js` file and display the alert box with the message "Hey everyone!".

Linking to a separate JavaScript file keeps the HTML and JavaScript code separate, which makes it easier to read and maintain. It also makes it easier to reuse the JavaScript code in multiple HTML files. For example, if we wanted to add the same alert to another HTML file, we could simply link to the same `js-in-html.js` file.

The `<script>` tag has other attributes that we can use to control how the JavaScript code is loaded and executed. Let's look at a few of them.

The `async` attribute

The `async` attribute tells the browser to load the JavaScript file asynchronously, which means that the browser can continue loading the HTML file while the JavaScript file is loading. This can be useful for large JavaScript files that take a long time to load. However, it can also cause problems if the JavaScript code depends on the HTML content being loaded first because, once the JavaScript file is loaded, the browser will execute the JavaScript code even if the HTML content hasn't finished loading yet. Here's how we would use the `async` attribute in a `<script>` element:

```
<script src="./js-in-html.js" async></script>
```

The `defer` attribute

The `defer` attribute tells the browser to load the JavaScript file asynchronously, but to execute the JavaScript code after the HTML file has been loaded. Unlike the `async` attribute, the `defer` attribute ensures that the JavaScript code is executed after the HTML content is loaded, avoiding any potential problems with the JavaScript code depending on the HTML content being loaded first. Here's how we would use the `defer` attribute:

```
<script src="./js-in-html.js" defer></script>
```

It's important to note that the `async` and `defer` attributes are not supported by all browsers, particularly older ones. However, they are a good option for controlling how your JavaScript code is loaded and executed.

Conclusion

We've seen the different ways to include JavaScript code in an HTML file. Each method has its advantages and disadvantages and the choice of which to use depends on various factors like the size and complexity of the project, the required level of compatibility with different browsers, and the need for optimization.

Regardless of which method you use, it's essential to keep your code organized and maintainable, so that you can create reliable web applications that provide an excellent user experience.

Browser Events

Browser events are actions that occur in the browser. These actions are usually initiated by the user, such as clicking on a button, scrolling through a web page or submitting a form. By using JavaScript, we can respond to these events and create dynamic, interactive web pages.

When a user clicks a button on your webpage, you might want to react to that action by displaying a message or navigating to a new page. The browser provides a way to listen for these events and respond to them. Through the DOM, we can add event listeners to elements in our HTML page so that when a user performs a specific action, the browser will execute a JavaScript function in response to it. This function is called an event handler. Event handlers are often referred to as event listeners, but more specifically, the event listener is what listens for the event while the event handler is the function that is executed when the event occurs.

DOM elements have an `addEventListener()` method that allows us to add event listeners to them. This method takes two arguments: the name of the event and the event handler function. For example, if we wanted to add a click event listener to a button, we would call its `addEventListener()` method and pass in the event name and the event handler function as arguments:

```
const button = document.querySelector('button')

button.addEventListener('click', () => {
  console.log('You clicked me!')
})
```

Apart from the `addEventListener()` method, elements also have several event properties that we can set to event handler functions.

These properties are named after the event they listen for, such as `onclick` for the `click` event or `onscroll` for the `scroll` event. For example, we can also add a click event listener to the button by setting its `onclick` property to an event handler function:

```
button.onclick = () => {
  console.log('You clicked me!')
}
```

When an event handler function is executed, the browser passes an `event` object as an argument to the function. This object contains information about the event that occurred. For example, the `event` object passed into a `click` event function contains information about the element that was clicked on. We can access this information by using the `target` property of the `event` object. For example, if we wanted to log the element that was clicked on, we could do the following:

```
button.onclick = (event) => {
  console.log(event.target)
}
```

Let's work with another HTML file so that you can see how event listeners work in practice. Create a new file named `browserEvents.html` and add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Browser Events</title>
  </head>
  <body>
    <h1>Working with Browser Events</h1>
    <p>
      Click the button below to trigger a browser
      event.
    </p>
    <button id="click-me-btn">Click Me!</button>
  </body>
</html>
```

Next, create a new file in the same directory named `browserEvents.js` and add the following JavaScript code to it. We'll start by selecting the button element and adding a click event listener to it:

```
const button = document.querySelector('#click-me-btn')

button.addEventListener('click', () => {
  alert('You clicked me!')
})
```

In our JavaScript file, we used the `querySelector()` method to select the button element and store it in a variable named `button`. Then, we added a click event listener to the button by calling its `addEventListener()` method and passing in the event name and the event handler function as arguments. The event handler function is an anonymous arrow function that displays an alert message. Let's include our JavaScript file in our HTML code so that the browser can execute it. Add the following code to the `browserEvents.html` file:

```
<script src="browserEvents.js"></script>
```

You can now open the HTML file in your browser and click on the button. You should see an alert message that says "You clicked me!".

Event Object Properties and Methods

If we want to get more information about the event that occurred or the element that was clicked on, we can pass an `event` parameter into the event handler function. The parameter will represent the `event` object that the browser passes into the function when it is executed, which contains information about the event that occurred. Some of the common properties and methods of the `event` object include:

target

The `target` property returns the element that an event occurs on. For example, if we want to log the element that was clicked on to the console, we can edit the `click` event handler function in the `browserEvents.js` file to the following:

```
button.addEventListener('click', (event) => {
  console.log(event.target)
})
```

Reloading your page and clicking on the button should log the button element `<button id="click-me-btn">Click Me!</button>` to the console.

`type`

The `type` property returns the type of event that occurred, such as `click` or `submit`:

```
button.addEventListener('click', (event) => {
  console.log(event.type)
})
```

`currentTarget` and Event Bubbling

The `currentTarget` property returns the element that the event listener is attached to, not necessarily the element that the event occurs on. The difference between `target` and `currentTarget` is that `target` returns the element that was clicked on, while `currentTarget` returns the element that the event listener is attached to. For example, if we remove the `click` event listener from the button and add it to the `body` element instead, the `currentTarget` property will return the `body` element, even if we click on the button. This is called **event bubbling**. *Event bubbling is a mechanism that allows an event to propagate or "bubble" up to its parent elements.* Since the `body` element is the parent of the `button` element, the `click` event will bubble up to the `body` element where the event listener is attached. Replace the code in your `browserEvents.js` file with the following:

```
document.body.addEventListener('click', (event) => {
  console.log(event.currentTarget)
})
```

`stopPropagation()`

The `stopPropagation()` method prevents the event from bubbling up to parent elements. For example, if we want to prevent the `click`

event from bubbling up to the `body` element when we click on the button, we can add an event listener to the button and call the event object's `stopPropagation()` method in the event handler function. Along with the `body` event listener from the previous example, add the following code to your `browserEvents.js` file and reload the page:

```
const button = document.querySelector('#click-me-btn')

button.addEventListener('click', (event) => {
  event.stopPropagation()
})
```

Now, when you click on the button, the `click` event will not bubble up to the `body` element and the `body` event listener will not be executed. You can also add code to the button's event handler function that will run when the button is clicked:

```
button.addEventListener('click', (event) => {
  event.stopPropagation()
  alert('You clicked me!')
})
```

Without the `stopPropagation()` method, both the `body` event listener and the button's event listener would be executed when the button is clicked.

key

The `key` property returns the value of the key that was pressed or released when a keyboard event occurs. For example, if we want to log the value of the key that was pressed to the console, we can edit the body event handler function in the `browserEvents.js` file to the following:

```
document.body.addEventListener('keydown', (event) => {
  console.log(event.key)
})
```

Reloading your page and pressing a key on your keyboard should log the value of that key to the console.

preventDefault()

The `preventDefault()` method prevents the default behavior of an event from being executed. For example, the default behavior of a form's submit button is to reload the page when it is clicked. If we want to prevent this behavior, we can call the `preventDefault()` method on the `event` object in the `submit` event handler function. Let's add a form to our `browserEvents.html` file. Add the following code under the button element:

```
<form id="name-form">
  <label for="name">Name:</label>
  <input type="text" id="name">
  <input type="submit" value="Submit">
</form>
```

When you click on the submit button in your browser, notice that the page reloads. Let's add an event listener to the form's submit button to prevent this behavior. Replace the code in your `browserEvents.js` file with the following:

```
const form = document.querySelector('#name-form')

form.addEventListener('submit', (event) => {
  event.preventDefault()
})
```

Now, clicking on the submit button should not reload the page. We can also add some code to the event handler function to log the value of the input field to the console:

```
form.addEventListener('submit', (event) => {
  event.preventDefault()
  console.log(event.target.name.value)
})
```

Reloading your page and submitting the form should log the value of the input field to the console. The `event.target` property returns the form element, the `name` property returns the input element, and the `value` property returns the value of the input field.

Browser Event types and listeners

Let's explore the different types of browser events and how we can use listeners to respond to them.

Mouse Events

Mouse events are actions that occur when a user interacts with the mouse, such as moving it or clicking on an element. The following are some of the most common mouse events:

- `click` - occurs when a user clicks on an element
- `dblclick` - occurs when a user double-clicks on an element
- `mousedown` - occurs when a user presses down on a mouse button while over an element
- `mouseup` - occurs when a user releases a mouse button while over an element
- `mouseover` - occurs when a user moves the mouse over an element

- `mouseout` - occurs when a user moves the mouse off of an element
- `mousemove` - occurs when a user moves the mouse while over an element
- `wheel` - occurs when a user scrolls the mouse wheel while over an element

Keyboard Events

Keyboard events are triggered by actions performed with the keyboard, such as typing and pressing keys. Some common keyboard events include:

- `keydown` - triggered when a key is pressed down
- `keyup` - triggered when a key is released
- `keypress` - triggered when a key is pressed and released

Form Events

Form events are triggered when a user interacts with a form, such as submitting it. Some common form events include:

- `submit` - triggered when a form is submitted
- `reset` - triggered when a form is reset

Window Events

Window events are triggered when a user interacts with the browser window, such as resizing it or scrolling through a page. Some common window events include:

- `load` - triggered when a page is loaded
- `unload` - triggered when a page is unloaded
- `resize` - triggered when the browser window is resized
- `scroll` - triggered when the user scrolls through a page

To use these events, we can add event listeners to the `window` object. *The `window` object represents the browser window and is the global object in a browser.* The `document` object, on the other hand, represents the HTML document and is a property of the `window` object. So you can access the `document` object by using `window.document` or just `document`. The `alert()` function is also a property of the `window` object, so you can access it by using `window.alert()` instead of just `alert()`. The window object has other properties and methods like the `load` property and the `addEventListener()` method. For example, if we want to log a message to the console when the page is loaded, we can add an event listener to the `window` object that listens for the `load` event. Add the following code to your `browserEvents.js` file, then reload the page:

```
window.addEventListener('load', () => {
  window.alert('The page has loaded!')
})
```

Now, when you reload the page, you should see an alert message that says "The page has loaded!". Let's try another example by logging the value of the `window.innerWidth` property to the console when the window is resized. The `window.innerWidth` property returns the width of the browser window in pixels. In your `browserEvents.js` file, add the following code:

```
window.addEventListener('resize', () => {
  console.log(window.innerWidth)
})
```

To resize your browser window, you can open your browser's developer tools and drag the right edge of the page to the left or right. You should see the value of the `window.innerWidth` property logged to the console as you resize the window. You can also log the value of the `window.innerHeight` property to the console to see the height of the browser window:

```
window.addEventListener('resize', () => {
  console.log(window.innerHeight)
})
```

You'll need to drag the bottom edge of the page up or down to see the value change. The resize event listener is useful for making your web pages responsive with JavaScript.

Let's try an example that logs the value of the `window.scrollY` and `window.scrollX` properties to the console when a user scrolls through the page. These properties allow you to determine the vertical and horizontal scroll positions of a web page. The `window.scrollY` property returns the vertical scroll position in pixels, and the `window.scrollX` property returns the horizontal scroll position in pixels. To add the scroll event listener, write the following code in your `browserEvents.js` file:

```
window.addEventListener('scroll', () => {
  console.log('v scroll position:', window.scrollY)
  console.log('h scroll position:', window.scrollX)
})
```

Since our current web page is not long enough to scroll, we can resize the browser window vertically to make it scrollable. With your browser's developer tools open, drag the bottom edge of the page up so that the page is longer than the browser window. Now, as you scroll through the page, you should see the vertical and horizontal scroll positions logged to the console.

Browser Storage

When building web applications, you'll often need to store data on the browser for various purposes like caching (storing data locally to improve performance), storing user preferences, or storing user data for easy retrieval. For this reason, modern browsers provide several mechanisms for client-side storage, including cookies, LocalStorage, and SessionStorage.

Cookies

Cookies are small pieces of data that are stored on a user's computer by the web browser. They were designed to be a reliable mechanism for websites to remember stateful information (such as items added to the shopping cart in an online store) or to record the user's browsing activity and store small amounts of data that could be used to persist information between page requests.

Cookies are typically stored in key-value pairs, and each cookie is associated with a specific domain. So a website can only access cookies it has set. Cookies are also limited in size, and can only store a maximum of 4KB of data. They can be set to expire at a specific time, after which they will be deleted by the browser. If a cookie does not have an expiration date, it is considered a session cookie and will be deleted when the user closes the browser. However, not all browsers automatically delete session cookies when the browser is closed, and there may be other factors that influence the behavior of cookies in the browser, such as user settings or extensions.

Since cookies are associated with a specific domain, you'll have to run your web page on a server to set and retrieve cookies. If you're working with VS Code or another extensive code editor, you can

install the **Live Server** extension or something similar to run your web page. In VS Code, for example, after installing the extension, right-click on any of your HTML files and select **Open with Live Server**. If not, you can use any other local server like the Node.js `http-server` package.

Storing Data in Cookies

With your web page running on a server, let's learn how to work with cookies by practice. We can set a cookie using the `document.cookie` property. This property is a string that contains all the cookies for the current domain. Let's set a cookie that expires on November 23, 2035:

```
document.cookie =
  'name=Ada Lovelace; expires=Fri, 23 Nov 2035 12:00:00 U\
TC; path=/'
```

In this example, the key-value pair `name=Ada Lovelace` is the actual data stored in the cookie, and the `expires` and `path` attributes are optional parameters that control the lifetime and accessibility of the cookie. The `expires` attribute specifies the expiration date of the cookie, and the `path` attribute specifies the path for which the cookie is valid. If the `path` attribute is not specified, the cookie will be valid for the entire domain.

To add a new cookie, we can set the `document.cookie` property again. This will append the new cookie to the existing string of cookies. Let's add a new cookie with the key-value pair `favorite_language=JavaScript`:

```
document.cookie = 'favorite_language=JavaScript'

console.log(document.cookie)
// "name=Ada Lovelace; favorite_language=JavaScript"
```

Retrieving Data from Cookies

We can retrieve the value of a cookie by using the `document.cookie` property. However, this property returns a string of all the cookies for the current domain, so we'll need to split the string into an array of individual cookies using the `split()` method. The `split()` method is a string method that takes a separator as an argument and returns an array of substrings. The separator is used to determine where each substring begins and ends. Since the cookies are separated by a semicolon and a space, we'll use "; " as the separator:

```
const cookies = document.cookie.split('; ')

console.log(cookies)
//["name=Ada Lovelace", "favorite_language=JavaScript"]
```

Let's retrieve the value of the `name` cookie. We can't always be sure of the order in which the cookies are stored, so we'll need to iterate through the array of cookies and check each one to see if it starts with the string `name`. If it does, we'll return that cookie. We can use the `find()` method to do this:

```
const nameCookie = cookies.find((cookie) =>
  cookie.startsWith('name')
)

console.log(nameCookie)
// output: "name=Ada Lovelace"
```

Remember that cookies have a limited size of 4KB, so they're not suitable for storing large amounts of data. Cookies are also sent to the server with every request, so they can slow down the performance of your web application if there are too many of them. So when working with cookies, it's best to store small amounts of data that are necessary for the application to function. If you need to store a larger amount of data, you can consider using LocalStorage or SessionStorage.

LocalStorage and SessionStorage

LocalStorage and SessionStorage are more recent additions to the browser storage options, and they offer several advantages over cookies. Like cookies, they allow web applications to store data on the client, but they have a larger storage capacity (typically around 5MB) and they don't send the data to the server with every request like cookies do.

LocalStorage and SessionStorage are similar in that they both store data on the client, but they differ in how long the data is stored. LocalStorage stores data with no expiration date, so the data will persist even after the browser window is closed. SessionStorage, on the other hand, stores data for only one session. This means that the data is deleted when the user closes the browser window or tab.

Let's see how we can use LocalStorage and SessionStorage to store data. We'll start with LocalStorage, then we'll move on to SessionStorage.

LocalStorage

LocalStorage is a property of the `window` object, so we can access it using `window.localStorage`. The `localStorage` property is an object that contains a `setItem()` method for setting data and a

`getItem()` method for retrieving data. Let's store the username for a user in LocalStorage. We'll use the `localStorage.setItem()` method for this. This method takes two arguments: the key and the value. The key is a string that will be used to retrieve the value. The value can be any data type, but it will be converted to a string when stored in LocalStorage and can be converted back to its original state when retrieved.

Let's store the username `ada_lovelace`:

```
localStorage.setItem('username', 'ada_lovelace')
```

To retrieve the value of the `username` key, we can use the `localStorage.getItem()` method. This method takes a key as an argument and returns the value associated with that key. Let's retrieve the value of the `username` key:

```
const username = localStorage.getItem('username')

console.log(username)
// output: "ada_lovelace"
```

We can also log the `localStorage` object to see all the keys and values stored in LocalStorage:

```
console.log(localStorage)
// output: Storage {username: "ada_lovelace"}
```

Storing and Retrieving Arrays and Objects

Arrays are automatically converted to strings when stored in LocalStorage, so we can store an array of usernames:

```
const allUsernames = [
  'ada_lovelace',
  'grace_hopper',
  'alan_turing',
]
localStorage.setItem('allUsernames', allUsernames)

const storedUsernames = localStorage.getItem(
  'allUsernames'
)

console.log(storedUsernames)
// output: "ada_lovelace,grace_hopper,alan_turing"
```

To convert the string back to an array, we can use the `split()` method:

```
const usernamesArray = storedUsernames.split(',')

console.log(usernamesArray)
// ["ada_lovelace", "grace_hopper", "alan_turing"]
```

Objects on the other hand need to be first converted to strings before being stored in LocalStorage. If you try to store an object directly, it will be automatically converted to the string `[object Object]` instead of the actual object data. This is because JavaScript's default `toString()` method returns the string `[object Object]` when called on an object.

To convert an object to a string, we can use the `JSON.stringify()` method. This method takes an object as an argument and returns a string representation of that object. Let's store an object in LocalStorage:

```
const user = {
  username: 'ada_lovelace',
  favoriteLanguage: 'JavaScript',
}

localStorage.setItem('user', JSON.stringify(user))
```

To retrieve the object from LocalStorage, we can use the `localStorage.getItem()` method. However, this method returns a string, so we'll need to convert it back to an object using the `JSON.parse()` method. This method takes a string as an argument and returns the object that the string represents:

```
const storedUser = localStorage.getItem('user')

console.log(storedUser)
// output: "{"username":"ada_lovelace",
// "favoriteLanguage":"JavaScript"}"

const userObject = JSON.parse(storedUser)

console.log(userObject)
// output: {username: "ada_lovelace",
// favoriteLanguage: "JavaScript"}
```

Removing Items from LocalStorage

To remove an item from LocalStorage, we can use the `localStorage.removeItem()` method. This method takes a key as an argument and removes the key-value pair from LocalStorage:

```
localStorage.removeItem('username')

console.log(localStorage.getItem('username'))

// output: null
```

Clearing LocalStorage

To clear all items from LocalStorage, we can use the `localStorage.clear()` method:

```
localStorage.clear()

console.log(localStorage)
// output: Storage {}
```

When working with LocalStorage, remember that the data is stored on the client and can be accessed by anyone or any script that has access to the client. So you should avoid storing sensitive data in LocalStorage, like passwords or credit card information. LocalStorage is also shared across all browser tabs and windows. This means that if you store data in localStorage in one tab, it will be available in all other tabs and windows that share the same origin (the same domain, protocol, and port). This can be useful for maintaining data across multiple tabs or windows, but it can also lead to conflicting or inconsistent states if you're not careful.

SessionStorage

The SessionStorage syntax is similar to LocalStorage. We can access it using the `window.sessionStorage` property. This property is an object that contains a `setItem()` method for setting data and a `getItem()` method for retrieving data. Let's use the same example as before to store the username `ada_lovelace` in SessionStorage:

```
sessionStorage.setItem('username', 'ada_lovelace')
```

To retrieve the value of the `username` key, we can use the `sessionStorage.getItem()` method:

```
const username = sessionStorage.getItem('username')

console.log(username)
// output: "ada_lovelace"
```

Like LocalStorage, we can remove items from SessionStorage using its `removeItem()` method:

```
sessionStorage.removeItem('username')

console.log(sessionStorage.getItem('username'))
// output: null
```

We can also clear all items from SessionStorage using its `clear()` method:

```
sessionStorage.clear()

console.log(sessionStorage)
// output: Storage {}
```

Remember that SessionStorage is only available for the current session. So if you close the browser window or tab, the data will be deleted. Unlike LocalStorage, SessionStorage is specific to a single browser tab; when you store data in SessionStorage in one tab, it will not be available in other tabs or windows and it will be deleted when the tab is closed.

Building A Todo List App with JavaScript

Let's use our knowledge of JavaScript and the DOM to build a simple to-do list app. In our app, we'll be able to add tasks to a list, mark them as completed, remove them from the list, and store them in the browser's **LocalStorage**, a web storage API that allows us to persist data in the browser so that even after we refresh the page or close the browser, our tasks will still be available.

Using HTML to structure our app

We'll start by creating an HTML file named `todoApp.html` and adding the following code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0"
    />
    <title>Todo App</title>
  </head>
  <body>
    <h1>Todo App</h1>
    <form id="addTaskForm">
      <input type="text" id="taskInput" required />
      <button type="submit">Add Task</button>
```

```
    </form>
    <ul id="taskList"></ul>
    <script src="./todoApp.js"></script>
  </body>
</html>
```

The HTML code creates a simple form with text input and a submit button. We've given the form an id of `addTaskForm`, and the text input an id of `taskInput` so that we can access them in our JavaScript code. We also have an empty unordered list with an id of `taskList` where we'll display our tasks. When a user types a task in the input field and clicks the submit button, we'll use JavaScript to create a new list item and add it to the unordered list. We'll also add a button to each list item that we'll use to remove any task from the list, and a checkbox for marking a task as completed. Before the closing `</body>` tag, we've added a script tag that will load our JavaScript code from the `todoApp.js` file.

Using JavaScript to add functionality to our app

Let's create our `todoApp.js` file in the same directory as our `todoApp.html` file. We'll start by getting references to the form, input field, and unordered list elements in our HTML file using the `document.querySelector()` method:

```
const taskInput = document.querySelector('#taskInput')
const taskList = document.querySelector('#taskList')
const addTaskForm = document.querySelector(
  '#addTaskForm'
)
```

Since we're not working with a `CSS` file, we'll use JavaScript to style our task list. Let's add the following code to the `todoApp.js` file. This

will give our list a margin of `1rem` at the top, remove the default bullet points, and increase the font size to `1.5rem` so that our tasks are easier to read. To add multiple styles to an element, we can set the element's `style` property to a string containing the styles we want to apply:

```
taskList.style = `
  list-style: none;
  margin-top: 1rem;
  font-size: 1.5rem;
`
```

Next, we'll create a function named `createTaskItem()` that will take the task input value as an argument and return a string containing the HTML code for a list item with a checkbox, label, and remove button:

```
const createTaskItem = (task) => `
<li>
  <input type="checkbox" name="task" value="${task}" />
  <label for="task">${task}</label>
  <button type="button">X</button>
</li>
`
```

Notice that we're directly returning the HTML code as a string. We could have used the `document.createElement()` method to create the elements and append them to the list item, but this is more straightforward and easier to read. Since we're returning a multi-line string, we've used the backtick character (‘) instead of the single quote character (’) to wrap the string. We're also using the task input value as the value and label for the checkbox, which will allow us to easily get the task value when we want to remove it from the list.

Rendering saved tasks to the browser

When a user loads the page, we want to render any tasks that were saved in the browser's LocalStorage. We'll start by creating a variable named `storedTasks` that will hold the tasks we'll get from LocalStorage. Since the tasks are stored as a JSON string, we'll use the `JSON.parse()` method to convert it to a JavaScript object. If there are no tasks in LocalStorage, we'll set the `storedTasks` variable to an empty array:

```
const storedTasks =
  JSON.parse(localStorage.getItem('tasks')) || []
```

When adding a task to the list, we'll also add it to the `storedTasks` array, and then use the `JSON.stringify()` method to convert the array back to a JSON string before storing it in LocalStorage with the key `tasks`. This is necessary because LocalStorage only supports storing strings. The OR operator (`||`) will return the value of the first operand if it's truthy, otherwise, it'll return the value of the second operand. In this case, if the value of `localStorage.getItem('tasks')` is `null`, the OR operator will return an empty array.

Next, let's create a function named `renderTasks()` that will render the tasks we have in LocalStorage to the browser. We'll use the `forEach()` method to loop through the `storedTasks` array and call the `createTaskItem()` function for each task. We'll then use the `insertAdjacentHTML()` method to add the returned HTML code to the unordered list:

```
const renderTasks = () => {
  storedTasks.forEach((task) => {
    taskList.insertAdjacentHTML(
      'beforeend',
      createTaskItem(task)
    )
  })
}
```

The `insertAdjacentHTML()` method takes two arguments: the position where we want to insert the HTML code, and the HTML code itself. In this case, we're using the `beforeend` position, which will add the HTML code at the end of the unordered list. Next, we'll call the `renderTasks()` function on page load. This will render the tasks we have in LocalStorage to the browser whenever the page loads:

```
window.onload = renderTasks
```

Our `todoApp.js` file should now look like this:

```
const taskInput = document.querySelector('#taskInput')
const taskList = document.querySelector('#taskList')
const addTaskForm = document.querySelector(
  '#addTaskForm'
)

taskList.style = `
  list-style: none;
  margin-top: 1rem;
  font-size: 1.5rem;
`

const createTaskItem = (task) => `
  <li>
    <input type="checkbox" name="task" value="${task}">
```

```
    <label for="task">${task}</label>
    <button type="button">
      X
    </button>
  </li>
`

const storedTasks =
  JSON.parse(localStorage.getItem('tasks')) || []

const renderTasks = () => {
  storedTasks.forEach((task) => {
    taskList.insertAdjacentHTML(
      'beforeend',
      createTaskItem(task)
    )
  })
}

window.onload = renderTasks
```

Adding tasks to the list

Now, let's create the function that will add a task to the list when the user clicks the submit button. We'll use the `addEventListener()` method to listen for the `submit` event on the form. When the event is triggered, we'll prevent the default behavior of the form, which is to reload the page. We'll then get the value of the input field, create a new list item using the `createTaskItem()` function, and add it to the `ul` element on the page. We'll also add the task to the `storedTasks` array and store it in LocalStorage with the key `tasks`, then we'll clear the input field by resetting the form using the `reset()` method which is available on all form elements:

```
const addTask = (event) => {
  event.preventDefault()

  const task = taskInput.value
  const taskItem = createTaskItem(task)
  taskList.insertAdjacentHTML('beforeend', taskItem)

  storedTasks.push(task)
  localStorage.setItem(
    'tasks',
    JSON.stringify(storedTasks)
  )

  addTaskForm.reset()
}
```

Next, we'll add an event listener to the form that will call the `addTask()` function when the form is submitted:

```
addTaskForm.addEventListener('submit', addTask)
```

We can also add the event listener using the `onsubmit` attribute on the form element in the `todoApp.html` file:

```
<form
  id="addTaskForm"
  onsubmit="addTask(event)"
></form>
```

This will do the same thing as the `addEventListener()` method, but we'll go with the `addEventListener()` method so that we can keep all of our JavaScript code in the `todoApp.js` file.

To test our progress so far, open the `todoApp.html` file in your browser. You should see the form and the unordered list. Enter a

task in the input field and click the submit button. You should see the task added to the list. Also, when you refresh the page, the task should still be there.

Marking tasks as complete

Now, let's add the functionality for marking tasks as complete. When a user ticks the checkbox for a task, we'll cross it out using the `textDecoration` style, and if they untick the checkbox, we'll undo the cross-out. Let's start by creating a function named `toggleTaskCompletion()`. Our function will take the event object as an argument, and we'll use the `target` property of the event object to get the checkbox that was clicked. Then we'll use the `parentElement` property to get the list item that contains the checkbox. Next, we'll use the `querySelector()` method to get the `label` element inside the list item. If the checkbox is checked, we'll set the `textDecoration` style to `line-through`, otherwise, we'll set it to `none`:

```
const toggleTaskCompletion = (event) => {
  const taskItem = event.target.parentElement
  const task = taskItem.querySelector('label')

  if (event.target.checked) {
    task.style.textDecoration = 'line-through'
  } else {
    task.style.textDecoration = 'none'
  }
}
```

Next, we'll add an onChange event listener to the checkbox input element in the `createTaskItem()` function. This will call the `toggleTaskCompletion()` function whenever the checkbox is

checked or unchecked. Update your `createTaskItem()` function to this:

```
const createTaskItem = (task) => `
  <li>
    <input type="checkbox" name="task" value="${task}"
      onChange="toggleTaskCompletion(event)"
    >
    <label for="task">${task}</label>
    <button type="button">
      X
    </button>
  </li>
`
```

Now, you can mark tasks as complete by ticking the checkbox. Since we're not storing the state of the checkbox in LocalStorage, refreshing the page will reset the checkbox to its default state.

Removing Tasks

Next, we'll add the functionality for removing tasks from the list. When a user clicks the `X` button, we'll remove the task from the `ul` list on the page, and from the `storedTasks` array, then we'll update the LocalStorage so that refreshing the page won't bring back the removed task. We'll start by creating a function named `removeTask()`. Our function will take the event object as an argument, and we'll use the `target` property of the event object to get the button that was clicked. Then we'll use the `parentElement` property to get the list item that contains the button.

```
const removeTask = (event) => {
  const taskItem = event.target.parentElement
}
```

Next, inside the `removeTask()` function, we'll use the `querySelector()` method to get the `label` element in the list item which contains the task text as its `innerText` property. Then we'll use the `indexOf()` method to get the index of the task in the `storedTasks` array:

```
const task = taskItem.querySelector('label').innerText
const indexOfTask = storedTasks.indexOf(task)
```

Finally, we'll use the `splice()` method to remove the task from the array, and the `setItem()` method to update the LocalStorage. Then we'll remove the list item from the page using the `remove()` method:

```
storedTasks.splice(indexOfTask, 1)
localStorage.setItem(
  'tasks',
  JSON.stringify(storedTasks)
)

taskItem.remove()
```

Now, we'll add an onClick event listener to the `X` button in the `createTaskItem()` function. This will call the `removeTask()` function whenever the `X` button is clicked. Update your `createTaskItem()` function to this:

```
const createTaskItem = (task) => `
  <li>
    <input type="checkbox" name="task" value="${task}"
      onChange="toggleTaskCompletion(event)"
    >
    <label for="task">${task}</label>
    <button type="button" onClick="removeTask(event)">
      X
    </button>
  </li>
`
```

You can now remove tasks from the list by clicking the `X` button. Your `todoApp.js` file should look like this:

```
const taskInput = document.querySelector('#taskInput')
const taskList = document.querySelector('#taskList')
const addTaskForm = document.querySelector(
  '#addTaskForm'
)

taskList.style = `
  list-style: none;
  margin-top: 1rem;
  font-size: 1.5rem;
`

const createTaskItem = (task) => `
  <li>
    <input type="checkbox" name="task" value="${task}"
      onchange="toggleTaskCompletion(event)"
    >
    <label for="task">${task}</label>
    <button type="button" onclick="removeTask(event)">
      X
```

```
      </button>
    </li>
  `

  const storedTasks =
    JSON.parse(localStorage.getItem('tasks')) || []

  const renderTasks = () => {
    storedTasks.forEach((task) => {
      const taskItem = createTaskItem(task)
      taskList.insertAdjacentHTML('beforeend', taskItem)
    })
  }

  window.onload = renderTasks

  const addTask = (event) => {
    event.preventDefault()

    const task = taskInput.value
    const taskItem = createTaskItem(task)
    taskList.insertAdjacentHTML('beforeend', taskItem)

    storedTasks.push(task)
    localStorage.setItem(
      'tasks',
      JSON.stringify(storedTasks)
    )

    addTaskForm.reset()
  }

  addTaskForm.addEventListener('submit', addTask)

  const toggleTaskCompletion = (event) => {
```

```
  const taskItem = event.target.parentElement
  const task = taskItem.querySelector('label')

  if (event.target.checked) {
    task.style.textDecoration = 'line-through'
  } else {
    task.style.textDecoration = 'none'
  }
}

const removeTask = (event) => {
  const taskItem = event.target.parentElement
  const task =
    taskItem.querySelector('label').innerText

  const indexOfTask = storedTasks.indexOf(task)
  storedTasks.splice(indexOfTask, 1)
  localStorage.setItem(
    'tasks',
    JSON.stringify(storedTasks)
  )

  taskItem.remove()
}
```

Conclusion

Now, you know how to create a simple to-do list app using HTML, CSS, and JavaScript. You've gained practical experience with getting input from the user, adding and removing elements from the DOM, storing data in LocalStorage, and using event listeners to respond to user actions. This is a great foundation for building more complex web apps. To test your knowledge, try building this app again, but this time, on your own. You can also add more features

to the app, such as a button to clear all tasks, or a button to mark all tasks as complete. If you get stuck, you can always refer back to this guide.

Asynchronous JavaScript

What is Asynchronous JavaScript?

Asynchronous JavaScript is an essential concept in modern web development. In traditional JavaScript, code is executed in a synchronous manner, meaning that each line of code is linearly executed one after the other (from top to bottom). This is fine for simple programs, but it becomes a problem when working with long-running tasks like making network requests, performing complex calculations, or accessing a database or file system. If you were to execute these tasks synchronously, the main thread responsible for rendering the user interface and handling user interactions would be blocked from doing anything else until the task is complete, making your program appear to be frozen or unresponsive to the user.

This is where asynchronous JavaScript comes in, allowing you to execute long-running tasks without blocking the main UI (user interface) thread, so your application can continue to run and respond to user input. The term "asynchronous" refers to a programming pattern where multiple tasks can be executed simultaneously, without blocking each other. This is necessary for modern web applications, which often need to perform multiple tasks at the same time, such as making network requests, loading data in the background, and playing audio or video.

Promises

One way to manage asynchronous code in JavaScript is by using Promises. A Promise is an object that represents the eventual

completion or failure of an asynchronous operation. You can think of it as a placeholder for a value that will become available in the future. A Promise is in one of three possible states:

- **pending**: The initial state of a Promise, representing that the value is not yet available.
- **fulfilled**: The state of a Promise representing a successful operation, meaning that the value has become available.
- **rejected**: The state of a Promise representing a failed operation, meaning that an error occurred.

Promises are created using the `Promise` constructor, which takes a single function as an argument. This function is called the **executor function** and is responsible for starting the asynchronous operation. The executor function is passed two functions as arguments: `resolve` and `reject`. The `resolve` function is called when the asynchronous operation is successful, and the `reject` function is called when the operation fails. The `resolve` and `reject` functions are used to change the state of the Promise and can be called with a value that will be passed to any handlers registered with the Promise. Here's an example of a Promise that resolves with a string value of "Data retrieved successfully" after a 5 second delay:

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data retrieved successfully')
  }, 5000)
})
```

When you run this code, the `myPromise` variable will be assigned a Promise object in the pending state. The Promise will remain in the `pending` state until the executor function calls the `resolve` function, which will change the state of the Promise to fulfilled and pass the string value to any handlers registered with the Promise. In this

case, the Promise will be fulfilled after a 5-second delay. If you try to log the `myPromise` variable to the console before the Promise is fulfilled, you will see that it is still in the `pending` state:

```
console.log(myPromise)

// Output: Promise { <pending> }
```

If you want to handle the result of the Promise, you can register a handler using its `then` method. The `then` method takes a function as an argument, which is called when the Promise is fulfilled. The function will have access to the value passed to the `resolve` function as its first argument. Here's an example of how to register a handler with a Promise:

```
myPromise.then((value) => {
  console.log(value)
})
```

This should output the string value "Data retrieved successfully" to the console after a 5-second delay. Note that the Promise will not stop other tasks from being executed while it is pending. So if we were to log something else to the console after the Promise is created, it would be logged immediately, without waiting for the Promise to be fulfilled, and when the Promise is fulfilled, its value would be logged to the console. To try this out, run the following code:

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data retrieved successfully')
  }, 5000)
})

myPromise.then((value) => {
  console.log(value)
})

console.log('Hey there!')
```

This will first log "Hey there!" to the console even though it comes after the Promise in the code, and then after a 5-second delay, the string value "Data retrieved successfully" will be logged to the console. Here's what the output would look like:

```
Hey there!
Data retrieved successfully
```

If instead of the `resolve` function, we were to call the `reject` function in our Promise, it would be rejected and any error handler registered with the Promise would be passed the error value. We can register an error handler using the `catch` method. The `catch` method takes a function as an argument, which is called when the Promise is rejected. The function will have access to the value passed to the `reject` function as its first argument. When working with Promises, it's important to handle both the fulfilled and rejected states, so that if there's an error you can respond to it appropriately. Here's an example of a Promise that is rejected after a 5-second delay, and an error handler that logs the error to the console:

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('An error occurred')
  }, 5000)
})

myPromise
  .then((value) => {
    console.log(value)
  })
  .catch((error) => {
    console.log(error)
  })
```

We can also write a Promise as the body of a function so that it can be run at a later time or reused:

```
const doSomeMath = (a, b) => {
  return new Promise((resolve, reject) => {
    const result = a + b
    if (result > 10) {
      resolve(result)
    } else {
      reject('The result is too small')
    }
  })
}
```

In our example, the `doSomeMath` function takes two numbers as arguments and returns a Promise. The Promise will be fulfilled if the result of adding the two numbers is greater than 10, and rejected if it is less than 10. We can call the `doSomeMath` function like this:

```
doSomeMath(5, 20)
  .then((value) => {
    console.log(value)
  })
  .catch((error) => {
    console.log(error)
  })

// Output: 25
```

Let's call it again, but this time pass in two numbers that will cause the Promise to be rejected:

```
doSomeMath(5, 2)
  .then((value) => {
    console.log(value)
  })
  .catch((error) => {
    console.log(error)
  })

// Output: The result is too small
```

Promise Methods

There are a few other methods that we can use with Promises. These methods include: `Promise.all`, `Promise.race`, `Promise.resolve`, `Promise.reject`, and `finally`. We'll start with `Promise.all`.

Promise.all

The `Promise.all` method takes an array of Promises as an argument and returns a single Promise. The single Promise that is returned

will be fulfilled when all of the Promises in the array argument are fulfilled. If any of the Promises in the array argument are rejected, the single Promise that is returned will be rejected. Here's an example of how to use `Promise.all`:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 1 resolved')
  }, 5000)
})

let promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 2 resolved')
  }, 3000)
})

let promise3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 3 resolved')
  }, 1000)
})

Promise.all([promise1, promise2, promise3]).then(
  (values) => {
    console.log(values)
  }
)

// Output: ["Promise 3 resolved", "Promise 2 resolved", "\
Promise 1 resolved"]
```

If we were to reject one of the Promises in the array, the single Promise that is returned would be rejected. Let's rewrite `promise2` so that it is rejected instead of resolved:

```
let promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('Promise 2 rejected')
  }, 3000)
})

Promise.all([promise1, promise2, promise3])
  .then((values) => {
    console.log(values)
  })
  .catch((error) => {
    console.log(error)
  })

// Output: Promise 2 rejected
```

Promise.race

The `Promise.race` method takes an array of Promises as an argument and returns a single Promise. The single Promise that is returned will be fulfilled or rejected as soon as one of the Promises in the array argument is fulfilled or rejected. Think of it like a race where the first Promise to finish wins. Let's try an example:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 1 resolved')
  }, 5000)
})

let promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 2 resolved')
  }, 3000)
})
```

```
12
13 let promise3 = new Promise((resolve, reject) => {
14   setTimeout(() => {
15     resolve('Promise 3 resolved')
16   }, 1000)
17 })
18
19 Promise.race([promise1, promise2, promise3]).then(
20   (value) => {
21     console.log(value)
22   }
23 )
24
25 // Output: Promise 3 resolved
```

In our example, `promise1` will resolve in 5 seconds, `promise2` in 3 seconds, and `promise3` in 1 second. Since `promise3` is the first Promise to resolve, the single Promise that is returned will be fulfilled with the value "Promise 3 resolved".

Promise.resolve

The `Promise.resolve` method takes a value as an argument and returns a Promise that is fulfilled with the value that was passed to it. It is useful when you need to convert a value into a Promise. Here's an example:

```
let myPromise = Promise.resolve('Promise resolved')

myPromise.then((value) => {
  console.log(value)
})

// Output: Promise resolved
```

Promise.reject

The `Promise.reject` method takes a value as an argument and returns a Promise that is rejected with the value that was passed to it:

```
let myPromise = Promise.reject('Promise rejected')

myPromise.catch((error) => {
  console.log(error)
})

// Output: Promise rejected
```

finally

The `finally` method takes a function as an argument and returns a Promise. The function that is passed to `finally` will be executed when the Promise that `finally` is called on is either fulfilled or rejected. Here's an example:

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise resolved')
  }, 5000)
})

myPromise
  .then((value) => {
    console.log(value)
  })
  .finally(() => {
    console.log(
      'Promise has been fulfilled or rejected'
    )
  })

// Output: Promise resolved
// Output: Promise has been fulfilled or rejected
```

Making HTTP Requests with Promises and the Fetch API

Introduction

Making HTTP requests to retrieve data from a server is an essential part of web development. With the introduction of JavaScript features like Promises and the Fetch API, making HTTP requests has become much simpler and more efficient. The **Fetch API** is a JavaScript API that provides a simple and convenient way of making HTTP requests, and it is supported by all modern browsers. *An API (Application Programming Interface) is a set of programming instructions and standards for accessing a specific service or data source.* We can use the Fetch API to make GET (retrieve data),

POST (create data), PUT (update data), and DELETE (delete data) requests. When the Fetch API is used to make a request, it returns a Promise that resolves to a Response object, which contains the data that was retrieved from the server.

To illustrate how to make HTTP requests with the Fetch API, we'll create a **JSON** file that contains the data we want to retrieve. *JSON stands for JavaScript Object Notation and is a lightweight format for storing and transporting data.* JSON is often used when data is sent from a server to a web page, and the data structure is similar to a JavaScript object. Let's create a JSON file called `data.json` that contains an array of objects. JSON files end with the `.json` extension. In our JSON data, each object will represent a user and will have a `name` and `age` property:

```
[
  {
    "name": "John",
    "age": 30
  },
  {
    "name": "Sara",
    "age": 22
  },
  {
    "name": "Bill",
    "age": 40
  }
]
```

In a real-world application, we would retrieve data like this from a server using an API URL. However, for this example, we'll use a local JSON file so that we don't have to worry about the server being available when we make the request.

Unlike in typical JavaScript objects, the keys in a JSON object must be wrapped in double quotes. Now that we have our JSON data,

we can use the Fetch API to make a GET request that retrieves the data from the JSON file. In the same directory as the `data.json` file, create an HTML file called `fetchExample.html` and add the following code to it:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Fetch API</title>
  </head>
  <body>
    <script src="./fetchExample.js"></script>
  </body>
</html>
```

Next, create a JavaScript file in the same directory called `fetchExample.js`. In our JavaScript file, we'll use the `fetch()` method to make a GET request to the `data.json` file. The `fetch()` method takes a URL as its first argument and returns a Promise that resolves to a Response object:

```
let fetchRequest = fetch('./data.json')

console.log(fetchRequest)
```

When you open the `fetchExample.html` file in your browser, you'll see a `CORS` error in the console. *CORS stands for Cross-Origin Resource Sharing and is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.* To avoid this error, we'll need to run a local server. If you're working with VS Code, you can install the Live Server extension. Once you've installed the extension, right-click on the `fetchExample.html` file and select

"Open with Live Server". If not, you can use any other local server like the Node.js `http-server` package.

Once you've started the local server, you'll see that the `fetchRequest` variable contains a Promise object with the status "pending":

```
Promise {<pending>}
```

The returned Promise will resolve to a Response object when the request is complete. Let's use the `then()` method to handle the Promise when it resolves. We'll log the Response object to the console:

```
fetchRequest.then((response) => {
  console.log(response)
})
```

The resolved response should look like this:

```
Response {body: ReadableStream, status: 200, type: "cors"\
,...}
```

The `Response` object contains information about the response, including the status code, the type of response, and the body of the response. The `body` property contains a `ReadableStream` object that we can use to read the data that was returned from the server. We can use the `json()` method to convert the response data to a JavaScript object. The `json()` method returns a Promise that resolves to a JavaScript object, so we'll chain another `then()` method to handle the Promise when it resolves:

```
fetchRequest
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    console.log(data)
  })
```

The `data` variable will contain the JavaScript object that was returned from the server:

```
(3) [{...}, {...}, {...}]
0: {name: "John", age: 30}
1: {name: "Sara", age: 22}
2: {name: "Bill", age: 40}
length: 3
__proto__: Array(0)
```

Let's use the `data` variable to display the user data on the page. We'll create a `ul` element and add a `li` element for each user. Update the `fetchExample.js` file to look like this:

```
let fetchRequest = fetch('./data.json')

fetchRequest
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    let ul = document.createElement('ul')

    data.forEach((user) => {
      let li = document.createElement('li')
      li.textContent = `${user.name} is ${user.age}`
      ul.appendChild(li)
```

```
    })

    document.body.appendChild(ul)
  })
```

When you refresh the page, you should see the user data displayed on the page:

```
John is 30
Sara is 22
Bill is 40
```

Async/Await

The `async` and `await` keywords were introduced in ES2017 to make working with Promises easier. For example, we can simplify the Fetch request in the `fetchExample.js` file by using the `async` and `await` keywords:

```
let fetchUsers = async () => {
  let response = await fetch('./data.json')
  console.log(response)
}

fetchUsers()

// Response {body: ReadableStream, status: 200, type: "co\
rs",...}
```

The `async` keyword is used to define an asynchronous function, while the `await` keyword is used to wait for a Promise to resolve, and can only be used inside an `async` function. In our example, we started by defining an asynchronous arrow function called `fetchUsers()`. We can also use the `async` keyword with a regular function declaration:

```
async function fetchUsers() {
  // ...
}
```

Inside the `fetchUsers()` function, we used the `await` keyword to wait for the `fetch()` method to resolve. When we use the `await` keyword in a variable declaration, the variable will be assigned the resolved value of the Promise. In our example, the `response` variable will be assigned the resolved value of the `fetch()` Promise. We can then use the `await` keyword to wait for the `json()` method to resolve. We'll create a new variable named `data` and assign it the resolved value of the `json()` Promise. Inside the `fetchUsers()` function, we'll log the `data` variable to the console:

```
let response = await fetch('./data.json')
let data = await response.json()

console.log(data)
```

When you refresh the page, you'll see the same user data that we saw in the previous example:

```
(3) [{...}, {...}, {...}]
0: {name: "John", age: 30}
1: {name: "Sara", age: 22}
2: {name: "Bill", age: 40}
length: 3
__proto__: Array(0)
```

We can now use the `data` variable to display the user data on the page:

```
let fetchUsers = async () => {
  let response = await fetch('./data.json')
  let data = await response.json()

  let ul = document.createElement('ul')

  data.forEach((user) => {
    let li = document.createElement('li')
    li.textContent = `${user.name} is ${user.age}`
    ul.appendChild(li)
  })

  document.body.appendChild(ul)
}

fetchUsers()
```

When you refresh the page, you should see the user data displayed on the page:

```
John is 30
Sara is 22
Bill is 40
```

We can use the `try` and `catch` keywords to handle errors that occur in an `async` function. Let's update the `fetchUsers()` function to use the `try` and `catch` keywords. We'll wrap the code that we want to execute in a `try` block, and then use the `catch` keyword to handle any errors that occur in it:

```
let fetchUsers = async () => {
  try {
    let response = await fetch('./data.json')
    let data = await response.json()
  } catch (error) {
    // Handle the error
    console.log(error)
  }
}
```

This will catch any errors that occur in the `try` block, including errors that occur in the `fetch()` method and the `json()` method.

The `async` and `await` keywords help us write asynchronous code that looks more like synchronous code, avoiding the need to chain different `.then()` methods. However, the `async` and `await` keywords are just syntactic sugar (syntax that is added to make the code easier to read) for Promises, and they don't change the underlying mechanism of how Promises work.

We've successfully used the Fetch API to retrieve data from our `data.json` file. We can also make an HTTP request to a server in the same way. Let's update the `fetchUsers()` function to make an HTTP request to the GitHub API. We can get the data for any user with their username. We'll try an example with mine:

```
let fetchUsers = async () => {
  try {
    let response = await fetch(
      'https://api.github.com/users/ebenezerdon'
    )
    let data = await response.json()
  } catch (error) {
    // Handle the error
    console.log(error)
  }
}
```

When you refresh the page, you'll see the data for the GitHub user with the username `ebenezerdon`. If you have a GitHub account, you can use your username instead of mine to see your data. Replace `ebenezerdon` with your username in the `fetch()` method and you should see information about your account.

Now you know how to work with Promises and the Fetch API. Practice everything you've learned by building a simple app that retrieves and displays data from a server. You can do a Google search for "public APIs" to find another API you can use. Remember that mastering JavaScript is a process, and you'll need to practice a lot to become good at it.

Working with Date and Time

JavaScript provides several ways to work with dates and times, allowing us to create, manipulate, and format them. The built-in `Date` constructor is the most common way to achieve this. It provides a way to create a date object, get and set the date and time components, and format the date and time into a string.

Creating a Date Object

We can create a new `Date` object by calling the `Date` constructor with no arguments:

```
let currentDate = new Date()

console.log(currentDate)
// output: Sat May 19 2025 08:59:03 GMT+0100
```

To create a `Date` object with a specific date and time, we can pass a string to the `Date` constructor:

```
let christmas = new Date('December 25, 2025')

console.log(christmas)
// output: Thu Dec 25 2025 00:00:00 GMT+0100
```

The `Date` constructor accepts several formats for the date and time string, including the following:

- December 25, 2025
- December 25 2025
- Dec 25 2025
- 12/25/2025
- 12-25-2025
- 2025-12-25
- 2025/12/25
- 2025.12.25
- 2025 12 25
- 2025 Dec 25
- 2025 December 25

We can specify the time in the string as well:

```
let christmas = new Date('December 25, 2025 23:15:30')

console.log(christmas)
// Thu Dec 25 2025 23:15:30 GMT+0100
```

If the time is not specified, it will default to midnight. The time format is `hours:minutes:seconds`, and the hours are specified in a 24-hour clock. We can also add a time zone to the string:

```
let christmas = new Date(
  'December 25, 2025 23:15:30 GMT-0500'
)

console.log(christmas)
// Fri Dec 26 2025 05:15:30 GMT+0100 (West Africa Standar\
d Time)
```

The `Date` constructor can also accept multiple arguments, which represent the year, month, day, hour, minute, second, and millisecond, respectively:

```
let christmas = new Date(2025, 11, 25)

console.log(christmas)
// Thu Dec 25 2025 00:00:00 GMT+0100
```

The month is specified as a number, where `0` is January, `1` is February, and so on. The day is specified as a number, where `1` is the first day of the month. The hour, minute, second, and millisecond are specified as numbers, where `0` is the first hour, minute, second, or millisecond of the day.

Getting the Date and Time Components

Once we have the `Date` object, we can extract several components of the date and time using its properties and methods. The `Date` object has the following properties:

- `getFullYear()` - returns the year as a four-digit number
- `getMonth()` - returns the month as a number
- `getDate()` - returns the day of the month as a number
- `getDay()` - returns the day of the week as a number
- `getHours()` - returns the hour as a number
- `getMinutes()` - returns the minutes as a number
- `getSeconds()` - returns the seconds as a number
- `getMilliseconds()` - returns the milliseconds as a number
- `getTime()` - returns the number of milliseconds since January 1, 1970. January 1, 1970 is used as the reference point for the Date object in JavaScript because it was the starting point of the Unix epoch, which is a standardized system for representing dates and times in computer systems.

Let's try an example with the `Date` object we created earlier:

```
let christmas = new Date(
  'December 25, 2025 23:15:30 GMT-0500'
)

console.log(christmas.getFullYear())
// output: 2025

console.log(christmas.getMonth())
// output: 11

console.log(christmas.getDate())
// output: 25

console.log(christmas.getDay())
// output: 5

console.log(christmas.getHours())
// output: 5
```

The `getMonth()` method returns the month in the range of 0-11, with 0 representing January and 11 representing December. Similarly, the `getDay()` method returns the day of the week in the range of 0-6, with 0 representing Sunday and 6 representing Saturday.

We can also use the `toString()` method to convert the `Date` object to a string:

```
let christmas = new Date(
  'December 25, 2025 23:15:30 GMT-0500'
)

console.log(christmas.toString())
// output: 'Fri Dec 26 2025 05:15:30 GMT+0100'
```

Setting the Date and Time Components

Once we have extracted the date and time components from a Date object, we can manipulate the components to get a new Date object representing a different date and time. One way to do this is to set the individual components of the date and time using the methods and properties of the Date object. The `Date` object has the following methods for setting the date and time components:

- `setFullYear(year)` - sets the year
- `setMonth(month)` - sets the month
- `setDate(date)` - sets the day of the month
- `setHours(hours)` - sets the hour
- `setMinutes(minutes)` - sets the minutes
- `setSeconds(seconds)` - sets the seconds
- `setMilliseconds(milliseconds)` - sets the milliseconds

Let's try an example with the christmas `Date` object:

```
let christmas = new Date(
  'December 25, 2025 23:15:30 GMT-0500'
)

christmas.setFullYear(2026)

console.log(christmas)
// output: Sat Dec 26 2026 05:15:30 GMT+0100
```

Formatting Dates and Times

The Date object has several methods for formatting the date and time into a string, including:

- `toDateString()` - returns the date portion of the `Date` object as a string
- `toTimeString()` - returns the time portion of the `Date` object as a string
- `toLocaleDateString()` - returns the date portion of the `Date` object as a string, formatted according to the locale
- `toLocaleTimeString()` - returns the time portion of the `Date` object as a string, formatted according to the locale
- `toUTCString()` - returns the date and time portion of the `Date` object as a string, formatted according to the UTC time zone

Let's try another example with the christmas `Date` object:

```
let christmas = new Date(
  'December 25, 2025 23:15:30 GMT-0500'
)

console.log(christmas.toDateString())
// output: Fri Dec 26 2025

console.log(christmas.toTimeString())
// output: 05:15:30 GMT+0100

console.log(christmas.toLocaleDateString())
// output: 26/12/2025

console.log(christmas.toLocaleTimeString())
// output: 5:15:30

console.log(christmas.toUTCString())
// output: Fri, 26 Dec 2025 04:15:30 GMT
```

There are other methods available in the `Date` object to a string, including `toISOString()`, `toJSON()`, and `toGMTString()`. You can

find more information about these methods in the MDN documentation*. Understanding how to create Date objects, manipulate them, and handle time zones is essential for creating accurate and user-friendly web applications. For example, you might need to display the current date and time on a web page, or the date of a specific entry in a Todo list. Working with the `Date` object can be a bit tricky, but with practice, you will be able to do so with ease.

*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Final Thoughts

Congratulations! You have reached the end of "Simplified JavaScript for Very Important Programmers".

When I set out to write this book, I wanted to create a resource that was accessible and easy to understand, even for those with no prior programming experience. I believe I've achieved this goal, and I hope that the explanations and examples throughout this book have helped reinforce your learning and build your confidence as a JavaScript programmer.

Remember that practice is the key to mastering any programming language, so I encourage you to continue applying your knowledge and take on new challenges.

Build projects and always come back here when you get confused or need to refresh your knowledge of any concept. You can find project-based JavaScript tutorials on my YouTube channel by searching for "Ebenezer Don" on YouTube or by visiting www.youtube.com/ebenezerdon*.

One important skill you need to develop as a programmer is your ability to collaborate with others and stay accountable. NewDev (www.newdev.io†) is my learning platform and developer community built to help you learn to code and connect with other developers. Consider sharing your daily progress with the community, as this will help you stay motivated and keep you on track.

You should also check the MDN Documentation‡ for the latest JavaScript updates and features.

*https://www.youtube.com/c/ebenezerdon
†https://www.newdev.io
‡https://developer.mozilla.org/en-US/docs/Web/JavaScript

If you enjoyed this book, please leave a review on Amazon or any other platform where you purchased it. I'd also love to hear from you on Twitter and LinkedIn, so feel free to reach out and say hello.

All my social media links are available on my NewDev profile page: www.newdev.io/ebenezer*

Thank you for reading, and I look forward to hearing all about your accomplishments on Social Media or through email (ebenezerdon@newdev.io).

Happy coding!

*https://www.newdev.io/ebenezer