



# The Complete Path to JS Mastery

The only JS Ebook you'll ever need!

{JS} By JavaScript Mastery

# Chapters

00 **Introduction**

01 **Variables and Data Types**

02 **Operators and Equality:**

03 **Logic and Control Flow**

04 **Functions**

05 **Tricky Concepts**

06 **Strings in Detail**

07 **Arrays in Detail**

08 **Objects in Detail**

# Chapters

- 09 **Value vs Reference**
- 10 **Document Object Model – DOM**
- 11 **Classes, "new" and "this"**
- 12 **Asynchronous JavaScript**
- 13 **Modern JS from ES6 to ES2020**
- 14 **Additional Resources**

CHAPTER 00

# Introduction

# Intro to JavaScript

Welcome! In this introductory section, you'll learn everything there is to know about the most popular programming language in recent history! Before we start let's quickly learn what exactly JavaScript is and why you need to learn it if you want to be a professional developer.

## So, what is JavaScript?

JavaScript is the programming language of the Web. The overwhelming majority of modern websites use JavaScript, and all modern web browsers – on desktops, game consoles, tablets, and smartphones – include JavaScript interpreters, making it the most present programming language in history.

# Intro to JavaScript

It's purpose is to infuse websites and web applications with interactivity!

From simple features, such as neat visual effects or displays of the current time and date to more sophisticated uses such as data fetching, chatrooms, browser games, web servers & even the creation of entire interfaces for things such as social media networks.

It is because of this that JavaScript is one of three must learn technologies for any web developer, whether it's a profession or simply a hobby. You learn HTML to create the content of your website, CSS to give it a fitting presentation and JS to assign it behaviors and allow it to come to life!

# Intro to JavaScript

**So, what are some examples of  
JavaScript usage that I can see today?**

It would be harder to find a service that doesn't use it. Have you ever talked to anyone using the text and voice chat application Discord?

A lot of its amazing features, such as bots are programmed using JavaScript and its desktop application is made using a JavaScript runtime, with which they easily ported every feature of the web application straight to your computer!

Did you watch a show on Netflix recently? Several JavaScript frameworks were used to build the applications' interface and achieve quicker startup times!

# Intro to JavaScript

What about PayPal transactions? Did you know their site uses JavaScript for various components of its checkout system?

Ever visit an impressive portfolio, with images flying in and out of view based on how far down you scroll, elements gradually popping in and out as you make your way through the site? That was most likely done with JavaScript as well.

In fact, let's take this back even further. Have you ever used a gallery on a website before? One that overlaid images on your screen, allowing you to scroll through picture after picture?

I'm sure you can guess what technology helped develop that! It's everywhere!

# Intro to JavaScript

## So what is the conclusion?

Whether you're someone looking to gain knowledge that could land them a great job, a hobbyist looking for an environment in which to create an amusing project, an artist looking to enhance their presentation website or an entrepreneur prototyping that one web app that could finally push their business into the global public eye of today's competitive industry, simply gaining knowledge of this language will give you a better opportunity at perfecting anything you might aspire to in the world of web development.

You've come till here, which means that you're truly interested in what JavaScript has to offer.

# Intro to JavaScript

Most people are struggling with JavaScript because they wander around the internet, from one incomplete YouTube video to another, from one shallow blog post to another. They don't have a plan. And that's the main problem that The Complete Path to JavaScript Mastery solves.

It has a comprehensive curriculum that starts with fundamentals and gradually progresses to extremely complex topics.

This eBook is the one and only resource that you need to be a proficient JavaScript Developer. After finishing the course, you will be able to start mastering any of the most popular JavaScript frameworks and libraries, such as: ReactJS, AngularJS, VueJS and even NodeJS.

# Intro to JavaScript

There is one thing that all of these frameworks and libraries have in common, and that is JavaScript! I always tell my clients, to master the fundamentals of JavaScript, and that's going to open the doors to mastering everything else.

If you have any questions in regards to improving your JavaScript or Web Development skills, feel free to send a message in the discord server.

# Intro to the eBook

JavaScript is the future of the web. The use of JavaScript in the frontend has been reaching it's peak recently.

Nowadays, JavaScript has also paved the way into backend development with Node JS. That's why in this course we'll be diving deep into it to make sure you come out with a good understanding of how it works.

How is this eBook set up? It starts with basics like variables and data types and gradually moves to more complex topics.

There's a lot to cover, but it's been broken up into bite-sized section.

# Intro to the eBook

In each chapter: We'll introduce the topic briefly and provide you with a list of things you should pay attention.

You'll be asked to read the chapters, expand on the things you've just learned, solve quizzes and generally do everything you can to best consume the material.

This eBook doesn't have any prerequisites, but don't think it is easy because of that. It's going to teach you the basics as well as most advanced JavaScript topics.

So, let's get started 🔥

**CHAPTER 01**

# **Variables and Data Types**

# Variables

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of giving variables a descriptive name, so our programs can be understood more clearly by the reader and ourselves.

It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used through the entirety of your program.

Variables in JavaScript are containers which hold reusable data. In other words, they are units of storage, like some sort of box into which we can put data.

# Variables

Here are the following three simple steps to creating a variable:

- Create a variable with the appropriate name.
- Store your value in it.
- Retrieve and use the stored value from the variable.

The values that we store in our variables can come in the form of predefined Data Types.

The computer needs to know of which type is our value so it can manipulate it properly.

# Variables

So, how can we create a variable? We need a var keyword, a variable name and the value we want to assign to that variable:

```
var variableName = 'Hello, World!';
```

Just a note, semicolons are used to terminate a line in javascript. They are optional, but omitting them can lead to undesired consequences on rare occasions. So it's a good practice to always put them at the end!

# Variables

In earlier versions of JavaScript, variables were solely declared using the var keyword followed by the name of the variable and a semicolon. This is how we would do it.

```
var variableName = '';
```

After ES6 (a newer version of JavaScript) we now have two new ways to declare a variable: let and const.

We can take a look at both of them one by one.

# Variables

The variable type let shares lots of similarities with var but unlike var it has some scope constraints. The scope is out of "scope" of this introductory video but we will explain it in great detail in a later video! The only thing that you need to know right now is that let is the preferred way of creating variables in modern JavaScript.

```
let variableName = '';
```

Const is another variable type assigned to data whose value cannot and will not change throughout the script.

```
const variableName = '';
```

# Variables

The **variableName** is the name of the variable which you can freely choose.

What do you think, can we name our variables literally anything?

We have a few criteria when it comes to creating variable names, also known as identifiers.

We have a few rules when it comes to creating an identifier in JavaScript:

- The name of the identifier must be unique
- the name of the identifier should not be any reserved JavaScript keyword (for example, we cannot declare a variable like this: var let = 0;

# Variables

- the first character must be a letter, an underscore (\_), or a dollar sign (\$). Subsequent characters may be any letter or digit or an underscore or dollar sign.

To recap, there are three different ways to make (or declare) a variable: var, let and const.

Const when variable is going to be constant and let when we plan on changing it!

Let's move on to data types to see what kind of data can we store inside of variables!

# Data Types

As mentioned, we can store values in variables. And these values need to be in the form of one of the predefined data types.

The concept of a value is somewhat abstract, especially to someone doing programming for the first time.

there are a few "types" of values, called data types. Let's go through them one by one, and then we're going to explain each one in detail.

We can separate data types of current JavaScript Standard in two groups:

- **Primitive Data Types**
- **Non-Primitive Data Types**

# Data Types

## Primitive Data Types:

<b>Number</b>	5, 6.5, 7 etc
<b>String</b>	"Hello everyone" etc
<b>Boolean</b>	true or false
<b>Null</b>	represents null i.e. no value at all
<b>Undefined</b>	A variable that has not been assigned a value is undefined.
<b>Symbol</b>	used as an identifier for object properties.

# Data Types

## Non-Primitive Data Types:

<b>Object</b>	instance through which we can access members
<b>Array</b>	group of similar values
<b>RegExp</b>	represents regular expression

# Comments

A comment is text in the code which is not read while we're running the code. You should write comments to explain what your code does.

Comments make your code easier to read and understand. They can help you and others who read your code.

There are two types of comments: multi-line and single-line comments.

- [Multi-line comments](#)
- [Single-line comments](#)

# Comments

## Multi-Line Comments

To write a comment that stretches over more than one line, you can use a multi-line comment, starting with the

```
/* characters and ending with */
```

closing it into a neat little rectangle. Anything between these characters is not processed by the JavaScript interpreter.

Multi-line comments are often used for descriptions of how script works.

# Comments

## Single-Line Comments

In a single-line comment, anything that follows the two forward slash characters on that line will not be processed by the JavaScript interpreter.

```
// single line comment
```

Single line comments are often used for short descriptions of what the code is doing.

Good use of comments will help you if you come back to your code after several days or months. They also help those who are new to your code.

# Strings

‘String’ is a sequence of characters.

String is a data type used to represent text.

Strings are simply fields of text. To wrap these words, we use quotes. A string in JavaScript must be surrounded by quotes.

In JavaScript, there are 3 types of quotes: **Single quotes, Double quotes, Backticks**

```
const singleQuote = 'Hello, everyone!';
```

```
const doubleQuotes = "Hello, everyone!";
```

Double and single quotes are “simple” quotes.  
There’s practically no difference between them.

# Strings

Backticks are “extended functionality” quotes.  
They allow us to embed variables & expressions  
into a string by wrapping them in ``${...}``

```
const backticks = `Hello! ${2 + 2}`;
```

The expression inside ``${...}`` is evaluated and the result becomes a part of the string.

We can put anything in there: a variable like name or an arithmetical expression like  $1 + 2$  or something more complex.

It is important to note that we can inspect the type of each value by writing `typeof` before the value.

# Numbers

JavaScript is real friendly when it comes to numbers, because you don't have to specify the type of number.

We call this behavior untyped. JavaScript is untyped because determining whether a number is an integer or a decimal(float) is taken care of by the language's runtime environment.

For example, in traditional programming languages like C, we'd have declare the type of the number we'd like to use. Like this:

```
int wholeNumber = 5;  
float decimalNumber = 0.5;
```

# Numbers

In JavaScript, we can just say use plain old const or let and use any number we'd like:

```
const wholeNumber = 5;  
const decimalNumber = 0.5;
```

We just learned that the number type represents both integer and floating point numbers.

There are many operations for numbers, e.g.

- + Addition
- Subtraction
- \* Multiplication
- / Division

# Numbers

When we try to do some operations with values that are not numbers, most often, we will get NaN as a result. **NaN** simply means, not a number, it represents a computational error.

It is a result of an incorrect or an undefined mathematical operation.

```
alert("this is a string" * 3);
```

What do you think we would get if we do a `typeof` of NaN?

```
console.log(typeof NaN);
```

# Numbers

The type of **NaN**, which stands for Not a Number is, surprisingly, a **number**.

The reason for this is, in computing, NaN is actually technically a numeric data type.

However, it is a numeric data type whose value cannot be represented using actual numbers.

Don't overthink this too much, numbers are quite straight forward in JavaScript, let's move on to Booleans!

# Booleans

Boolean represents a logical entity and can have only two values: **true** or **false**.

As you'll come to know, these are important values when it comes to adding logic to our programs. With just those two values, you can create a complex system of loops & conditions.

This type is commonly used to store yes/no values:

- ▶ **true** means “yes, correct”
- ▶ **false** means “no, incorrect”

# Booleans

```
const isCool = true;

if(isCool) {
  console.log("Hi man, you're cool");
} else {
  console.log("Oh.. Hi..");
}
```

Boolean values also come as a result of comparisons.

```
const age = 20;

if (age >= 18) {
  console.log('You may enter');
}
```

# Null and Undefined

## NULL

This type has only one value: **null**.

The special null value does not belong to any of the types described above. It forms a separate type of its own which contains only the null value

```
let age = null;
```

**null** is just a special value which represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown or empty for some reason.

# Null and Undefined

## UNDEFINED

A variable that has not been assigned a value is **undefined**.

The special value `undefined` also stands apart. It makes a type of its own, just like `'null'`. The meaning of `undefined` is “value is not assigned”.

If a variable is declared, but not assigned, then its value is `undefined` by default:

```
let x;  
console.log(typeof NaN);
```

# Null and Undefined

Technically, it is possible to assign undefined to any variable:

```
let x = 123;  
x = undefined;  
alert(x); // "undefined"
```

But I wouldn't recommend doing that. Normally, we use null to assign an "empty" or "unknown" value to a variable.

And we use undefined for checks like seeing if a variable has been assigned.

# Null and Undefined

## UNDEFINED VS NULL

Many a times we often get confused on whats the difference between UNDEFINED and NULL.

**undefined** means a variable has been declared but has not yet been assigned a value. On the other hand, **null** is an assignment value. It can be assigned to a variable as a representation of no value. Also, undefined and null are two distinct types: undefined is a type itself (**undefined**) while null is an **object**.

Unassigned variables are initialised by JavaScript with a default value of undefined. JavaScript never sets a value to null. That must be done programmatically.

# Objects

Object is the most important data-type and forms the building block for modern JavaScript.

The **object** type is special. All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever).

What I'm going to let you know for now is that objects in their simplest forms are used to group variables.

For example, we can create a variable of name, and age:

```
const name = 'John';
const age = 25;
```

# Objects

These two variables in the current state are in no way related on to another.

We can create an object called person and put them together:

```
const person = {  
  name: 'John',  
  age: 25;  
}
```

Now we know that both name and age belong to the same entity, the person. That is an object.

As you can see, we declare it the same as all other variables and then put curly brackets inside of which goes the data.

# Objects

The one last thing that we can mention is that we can now extract specific values from that object using the dot notation:

```
person.name
```

```
person.age
```

There are many other kinds of objects in JavaScript:

▶ **Array** to store ordered data collections,

▶ **Date** to store the information about the date and time

▶ **Error** to store the information about an error.

...And so on.

# Objects

They have their special features that we'll study later. Sometimes people say something like "Array type" or "Date type", but formally they are not types of their own, but belong to a single "object" data type.

And they extend it in various ways.

That's all that I'm going to let you know for now. Objects are complex concepts. First, let's master the easy things, and then we can get back to them later.

# **Statically vs Dynamically Typed Languages**

There are two types of languages when it comes to data types:

**Statically typed language** is where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types. Example: C, C++, Java.

**Dynamically typed languages** can receive different data types over time. In this all type checks are performed in a runtime, only when your program is executing.

JavaScript is dynamically typed; variables in JS can receive different data types over time.

# Statically vs Dynamically Typed Languages

A variable in JavaScript can contain any data. A variable can at one moment be a string and at another be a number, for example:

```
let message = "Hello, World!";  
  
message = 123456;
```

... It's a completely valid syntax.

**CHAPTER 02**

# **Operators and Equality**

# Operators

We know many operators from school. They are things like addition **+**, multiplication **\***, subtraction **-**, and so on.

In this chapter, we'll concentrate on aspects of operators that are not covered in school.

As all other programming languages, JavaScript includes operators as well. An operator performs some operation on single or multiple operands (data value) and produces a result.

For example  $1 + 2$ , where **+** sign is an operator and 1 is left operand and 2 is right operand. **+** operator adds two numeric values and produces a result which is 3 in this case.

# Operators

JavaScript includes following categories of operators.

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Assignment Operators
5. Conditional Operators

In the next few pages, we're going to explore all of these operators in detail.

# Arithmetic Operators

Arithmetic operators are used to perform mathematical operations between numeric operands.

We know many of them from school. They are things like addition +, multiplication \*, subtraction -, and so on. Arithmetic operators are something that we've all used before.

If we'd want to perform a mathematical operation, we'd want to use arithmetic operators.

# Arithmetic Operators

Arithmetic operators perform arithmetic on numbers (literals or variables).

**+** Addition

**-** Subtraction

**\*** Multiplication

**/** Division

**( )** Grouping operator

**%** Modulus (remainder)

**++** Increment numbers

**--** Decrement numbers

# Arithmetic Operators

```
// addition, subtraction, multiplication  
5 + 5;  
5.5 - 5;  
5 * 2;  
  
// division, modulo operator  
11 / 2;  
11 % 2;  
  
// increment, decrement  
5++;  
4--;  
  
// exponentiation  
4 ** 2;
```

Pretty straight forward right? Let's move on to comparison operators!

# Comparison Operators and Equality

As you've already heard of arithmetic operators, the same goes for comparison operators. You've most likely heard of greater than `>`, less than `<` or equal too `==`.

Comparison Operators compare two values and return boolean value `true` or `false`.

That sentence is the main takeaway. If you understand that you're good to go!

*The return value of a comparison operator is ALWAYS going to be a boolean value.*

The topic of Equality in JavaScript is closely connected to the comparison operators - I've got you covered there as well.

# Comparison Operators and Equality

```
const a = 5;
const b = 10;

// we can test whether a value is greater than the other value
console.log(a > b);
// we can also test whether a value is greater than or equal to the other value
console.log(a >= b);
// looks like I have a font installed so for me, it's going to look like one sign,
// but it's two. The greater than sign, and then immediately the equality sign.

// we can test whether a value is less than the other value
console.log(a < b);
// we can also test whether a value is less than or equal to the other value
console.log(a <= b);

// finally, we have the equality operators,
// we can test whether a value is equal
console.log(a == b);
// we also test whether a value is not equal
console.log(a != b);

// what you're going to see more often are going to be the strict equality
// and strict inequality operators. They look like this.

// strict equal, strict not equal
console.log(a === b);
console.log(a !== b);
```

# Comparison Operators and Equality

Everything that we've covered is pretty straight forward. The only thing that I'd like to take a deeper look at is the strict vs loose equality. What are the differences and when should we use each option? Let's cover that next!

# Strict vs Loose Equality

Equality is a fundamental concept in JavaScript. We say two values are equal when they are the same value. For example:

```
console.log('JSM' === 'JSM'); // true  
console.log(2 === 2); // true
```

Note that we use three equal signs to represent this concept of equality in JavaScript.

JavaScript also allows us to test loose equality. It is written using two equal signs. Things may be considered loosely equal even if they refer to different values that look similar, an example would be the following:

```
console.log(5 == "5"); // true
```

# Strict equality using ===

The strict equality method of comparison is a preferred option to use because it's behaviour can be easily predicted, which leads to less bugs and unexpected results.

The JavaScript interpreter compares the values as well as their types and only returns true when both are the same.

```
console.log(20 === "20"); // false
```

The code above will print false because even though the values seem to be the same, they are of different types. The first one is of type String and the second is of type Number.

# Strict equality using ===

Here's just one short thing that I wanted to show you. If we strictly compare objects, we're never going to get true. Let's test it out:

```
console.log({} === {}); // false  
// we get false, even though they have  
the same type and content, weird
```

```
// the same thing happens for arrays as  
they are actually objects under the hood  
console.log([] === []); // false
```

# Strict equality using ===

For the sake of simplicity, we're not going to go into too much depth about non-primitive data types like objects and arrays. That is a rather complex topic on its own. Because of that, later in the ebook we have a whole separate section called **Value vs Reference**.

In there we're going to explore the mentioned inconsistencies of the equality operator.

Now let's move on to the loose equality.

# Loose equality

We write loose equality using double equal sign.

It uses the same underlying logic as the Strict equality except for a minor, yet huge, difference.

The loose equality doesn't compare the data types. You should almost never use the loose equality.

Douglas Crockford's in his excellent book called **JavaScript: The Good Parts** wrote:

JavaScript has two sets of equality operators:  
`==` and `!=`, and their evil twins `==` and `!=`.

The good ones work the way you would expect. If the two operands are of the same type and have the same value, then `==` produces true and `!=` produces false.

# Loose equality

The evil twins do the right thing when the operands are of the same type, but if they are of different types, they attempt to change the values. The rules by which they do that are complicated and unmemorable.

These are some of the interesting cases:

```
'' == '0'          // false
0 == ''            // true
0 == '0'           // true

false == 'false'   // false
false == '0'        // true

false == undefined // false
false == null       // false
null == undefined  // true
```

Here are a few more examples: →

# Loose equality

## Using the == operator

```
// true, because 'true' is converted to 1 and  
then compared  
  
true == 1;  
  
// true, because the string of "5" is converted  
to the number 5 and then compared  
  
5 == "5";
```

## Using the === operator

```
true === 1; // false  
  
5 === "5"; // false
```

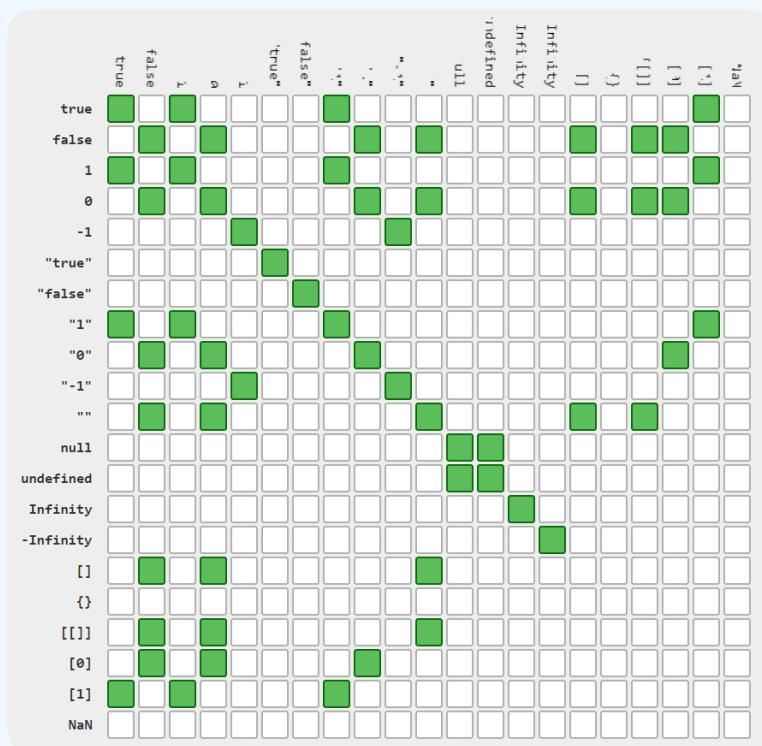
That's exactly how it should be. On the other hand

```
5 == "5"; // true
```

# Loose equality

This isn't and should never be equal. "5" is a string, and should be treated like that. As I mentioned, most of the JavaScript developers completely avoid loose equality and rely only on the strict equality. It is considered a better practice and it causes less bugs.

And for the end, I found for you a great visual representation of strict versus loose equalities:



[Link of the site →](#)

# Loose equality

As you can see, using the loose equality we get these green boxes all over the place. They're unpredictable. But if we switch to the strict equality, we get this nice predictable line.

So what's the moral of the story?



**Always use three equal signs.**

# Logical Operators Part 1

Logical operators are used to combine two or more conditions. JavaScript includes three logical operators:

**&&** AND

**||** OR

**!** NOT

Complete knowledge of logical operators requires the knowledge of if/else statements and truthy and falsy values.

For now we're just going to cover the syntax of logical operators and then we're going to come back to them to see them in full action after we cover the two mentioned topics!

# And Operator ( && )

Double ampersand `&&` is known as AND operator.  
It checks whether ALL OPERANDS are truthy values  
(we're going to explain truthy and falsy values in  
one of the later videos). And if they are truthy, it  
returns `true`, otherwise it returns `false`.

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

We can also pass multiple conditions:

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

# Or Operator ( || )

Double ampersand || is known as OR operator. It checks whether AT LEAST ONE OPERAND is a true value. And if there is at least one true, it returns true, otherwise it returns false.

```
console.log(true || false); // true  
console.log(true || true); // true  
console.log(false || false); // false
```

We can also pass multiple conditions:

```
console.log(true || true || false); // true
```

# Not Operator (!)

An exclamation sign ! is known as NOT operator.

It reverses the boolean result of the condition.

The syntax is pretty simple:

```
console.log(!true); // false  
console.log(!false); // true
```

As you can see, the not operator simply converts boolean false to true, and boolean true to false.

This was just an introduction to these logical operators. They are used really really often in real JavaScript applications and I'm excited to show you their real uses once we learn about if statements and truthy and falsy values!

# Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand.

Would you believe me if I told you that you not only know what an assignment operator is, but that you've been using it this whole time? The simplest form of an assignment operator is the = equal sign used for assigning values to variables:

```
const number = 5;
```

This right here is an assignment operator.

# Assignment Operators

We can also join the assignment operator with one of the arithmetic operators:

```
let number = 5;

number += 5; // the same as number = number + 5;
number -= 5; // the same as number = number - 5;
number *= 5; // the same as number = number * 5;
number /= 5; // the same as number = number / 5;

console.log(number);
```

The addition assignment can also be used with strings! Let me show you:

```
let string = 'Hello';
string += ', I am John.';
console.log(string); // "Hello, I am John."
```

That's it when it comes to assignment operators, you're basically a pro at them! :)

**CHAPTER 03**

# **Logic and Control Flow**

# Logic and Control Flow

You might have read the title of the current section, "Logic and Control Flow", and you might have wondered what does that mean. It's much simpler than what it may seem.

In every programming language, we have something known as an **if statement**.

If statement is consisted of a condition that is evaluated to either true or false and of a block of code. If the condition is evaluated to true, then the code inside of the block will run, otherwise, it's going to be skipped.

It's that simple. Let's explore it in an example:

# Logic and Control Flow

Let's think of where if statements appear in real life. They often appear in terms of some rules or laws.

Let's say that there's a night club that only allows people over the age of 18 to enter.

```
const age = 18;

if (age >= 18) {
    console.log('You may enter, welcome!');
}
```

# Logic and Control Flow

If statements can also have an **else if** and **else** statements. To continue with our example:

```
const age = 18;

if (age >= 18) {
    console.log('You may enter, welcome!');
} else if (age === 18) {
    console.log('You just turned 18, welcome!');
}
```

This is how we add an if else statement. Notice how we have another condition there.

If we run this code, what do you expect to see in your console?

Some of you might expect the code that's under the else if statement, but currently, that would not be the case, let's test it out.

# Logic and Control Flow

```
"You may enter, welcome!"
```

Console

Why did the first block got logged out? For a really simple reason. It was the first one to pass the check. Our first condition specified that the age must be more than or EQUAL TO 18.

18 is indeed equal to 18. So how could we fix this? We can simply exchange the more than and equal to to simply more than sign.

```
if (age > 18) {  
    console.log('You may enter, welcome!');  
} else if (age === 18) {  
    console.log('You just turned 18, welcome!');  
}
```

If we test it out again, now we get exactly what we expected. Awesome!

# Logic and Control Flow

But what if the person is younger than 18? We currently aren't handling that case. That's where the else statement comes in handy.

We implement it like this:

```
const age = 18;

if (age >= 18) {
    console.log('You may enter, welcome!');
} else if (age === 18) {
    console.log('You just turned 18, welcome!');
} else {
    console.log('Go away!');
}
```

Try noticing the difference between the if and the if else opposed to the else statement.

Can you see it?

# Truthy/Falsy Values

In the previous section, we learned about the strict and loose equality in JavaScript. Equality always results in a boolean value, it can either be a boolean **true** or a boolean **false**.

Unlike other languages, **true** and **false** values are not limited to boolean data types and comparisons. It can have many other forms.

In JavaScript, we also have something known as **truthy values** and **falsy values**.

Truthy expressions always evaluate to boolean **true** and falsy evaluate to boolean **false**.

# Truthy/Falsy Values

Knowledge of truthy and falsy values in JS is a must, if you don't know which values evaluate to truthy and which to falsy, you're going to have a hard time reading other people's code.

Longtime JavaScript developers often toss around the terms "truthy" and "falsy", but for those who are newer to JavaScript these terms can be a bit hard to understand.

When we say that a value is "truthy" in JavaScript, we don't just mean that the value is true.

Rather, what we mean is that the value coerces to true when evaluated in a boolean context.

Let's look at what that means.

# Truthy/Falsy Values

The easiest way to learn truthy and falsy values is to memorise falsy values only. There are only six falsy values, all the other values are truthy.

Let's explore the list of falsy values:

## FALSY Values

- false
- 0 (zero)
- "", ` `` (empty strings)
- null
- undefined
- NaN (not a number)

**Note :** Empty array ([]) is not falsy

# Truthy/Falsy Values

## TRUTHY Values

- Everything that is not FALSY

That's a pretty straightforward list.

Everything else is truthy. That also includes:

- '0' (a string containing a single zero)
- 'false' (a string containing the text "false")
- [] (an empty array)
- {} (an empty object)
- function(){} (an "empty" function)

A single value can therefore be used within conditions. For example: →

# Truthy/Falsy Values

```
if (value) {  
    // value is truthy  
} else {  
    // value is falsy  
    // it could be false, 0, '', null, undefined or NaN  
}
```

Taking advantage of truthiness can make your code a little bit more concise. We don't need to explicitly check for `undefined`, `""`, etc. Instead we can just check whether a value is truthy. However, there are some caveats to keep in mind.

I hope this introduction to truthiness and falsiness helps you to write clearer and more concise JavaScript.

# Logical Operators Part 2

And we're back to logical operators! Remember I've told you that we're going to return to them, I didn't forget!

Let's do a quick recap and then we're going to see some real and more complex examples of all three logical operators.

Logical operators are used to combine two or more conditions. If you remember correctly, JavaScript includes three logical operators:

 AND

 OR

 NOT

# And Operator ( && )

Double ampersand **&&** is known as AND operator. It checks whether two operands are truthy values. And if they are truthy, it returns **true**, otherwise it returns **false**. They are often used as a condition in an if statement.

So let's show that in an example.

Lets say that we want to choose which people may enter our club. To enter, they need to be cool, and they also need to be older than 18.

```
const age = 19; // age is represented by a number
const isCool = true; // isCool is represented by a boolean

if (isCool && age > 18) {
    // notice how we didn't say isCool === true
    console.log('You may enter.');
} else {
    console.log('You cannot enter.');
}
```

# And Operator ( && )

isCool is true, and age is greater than the 18, that means that we're going the if block of code is going to be executed.

Now, the point of this section is not an if/else statement. Let's remove that so we can focus purely on the logical operators. The only thing that we're going to keep is the condition. Let's log it to the console:

```
const age = 19;  
const isCool = true;  
  
console.log(isCool && age > 18); // true
```

We got true which is not a surprise. But now, instead of these true boolean values, let's test it with some truthy values.

# And Operator ( `&&` )

```
console.log('truthy' && 1 && 'test' && 999); //999
```

The output of this is 999, and why is that so? Shouldn't the logical operator AND return a boolean value? Here's how it works.

The AND `&&` operator does the following:

- It evaluates operands from left to right.
- It converts them to a boolean value.
  - If the result is true, it continues to the next value.
  - If the result is false, it stops and returns the original value of that operand.
- If all operands have been evaluated to true, it returns the last operand.

# And Operator ( && )

Now you know why 999 was returned, all the values were truthy and it was the last one on the list. Now what if we change one value to be falsy?

```
console.log('truthy' && 0 && 'test' && 999); //0
```

As you can see, if even one falsy value exists, it's going to stop and immediately return that value.

In other words, AND returns the first falsy value or the last truthy value if no falsy values have been found. Now let's move on to the OR operator.

# Or Operator ( || )

The syntax for the OR operator are two straight vertical lines **||**. It checks whether any one of the two operands is a truthy value.

Lets's see it in action:

```
console.log('truthy' || 0 || 'test' || 999); // "truthy"
```

We get **truthy**, why is that? Well let's see how the OR operator works:

The OR **||** operator does the following:

- For each operand, it converts it to boolean.
- If the result is true, it stops and returns the original value of that operand.
- If all operands have been evaluated to falsy returns the last operand.

# Or Operator ( || )

In other words, a chain of OR "||" returns the first truthy value or the last one if no truthy value is found. So now, if we change all of the values to be falsy, it is going to return the last one:

```
console.log('' || 0 || null || undefined); // undefined
```

As you can see, we get undefined.

# Not Operator (!)

An exclamation sign **!** is known as NOT operator.

It reverses the boolean result of the condition.

The syntax is pretty simple:

```
console.log(!true); // false
```

The operator accepts a single argument and does the following:

1. Converts the operand to boolean type:

**true/false.**

2. Returns the inverse value.

# Not Operator (!)

For instance:

```
alert(!true); // false  
alert(!10); // true
```

A double NOT !! is sometimes used for converting a value to boolean type:

```
alert (!!'truthy'); // true  
alert (!!null); // false
```

That is, the first NOT converts the value to boolean and returns the inverse, and the second NOT inverses it again. In the end, we have a plain value-to-boolean conversion.

# Not Operator (!)

This was a long section, so let's try to summarize it and repeat what we've learned.

JavaScript is lazy. It will want to do the least amount of work possible to get its return value.

With the AND operator: JavaScript will first try to return the first falsy value. If none were found, it will return the last truthy value.

And with the OR operator: JavaScript will first try to return the first truthy value. If none were found, it will return the last falsy value.

# Switch Statement

Switch statement is extremely similar to the if statement. They can be used interchangeably, but there are some situations where switch is preferred.

With if statements, you mostly have just a few conditions, one for the if, a few for the if else, and the final else statement.

If you have a larger number of conditions, you might want consider using the switch statement. Let's explore how it works.

The switch statement is used to perform different operations based on different conditions.

# Switch Statement

Let's say that you have a variable called

```
const superHero = 'Captain America';
```

Based on the name of the super hero, you want to display his voice line.

We can do that using the switch statement.

Switch statement takes in a value, and then checks it on bunch of cases:

```
switch (superHero) {
  case 'Iron Man':
    console.log('I am Iron Man... ');
    break;
  case 'Thor':
    console.log('That is my hammer! ');
    break;
  case 'Captain America':
    console.log('Never give up. ');
    break;
  case 'Black Widow':
    console.log('One shot, one kill. ');
    break;
}
```

# Switch Statement

Let's test it out to see what's the output and then we're going to explain it in more detail.

```
"Nerver give up."
```

Console

Great, we get exactly what we expected.

In this example, we passed the superHero variable to the switch statement. It executes the first check with triple equal sign.

It looks something like this:

```
'Captain America' === 'Iron Man'
```

Since this evaluates to false, it skips it and goes to the next one. As soon as it finds the one that matches correctly, it prints the output.

# Switch Statement

What is that break keyword? Break keyword ends the switch when we get the correct case.

If we omit the break statement, the next case will be executed even if the condition does not match the case.

And finally, what if none of the names in the switch match the name of our **superHero**, there must be something like an else statement, right? There is! That something is called **default**.

If none of the cases match, the default case is going to be executed.

We can implement it like this: →

# Switch Statement

```
switch (superHero) {  
    case 'Iron Man':  
        console.log('I am Iron Man...');  
        break;  
    case 'Thor':  
        console.log('That is my hammer!');  
        break;  
    case 'Captain America':  
        console.log('Never give up. ');  
        break;  
    case 'Black Widow':  
        console.log('One shot, one kill. ');  
        break;  
    default:  
        console.log('Enter a valid superhero name');  
        break;  
}
```

That's it! Now you know how to use the switch statement. As always, I would advise going into the chrome console and playing with it yourself.

**You learn the most by trying things yourself.**

# Ternary Operator

You could say that the switch statement is a bit more complicated version of the if statement. There's yet another version of it. It's called the ternary operator. It should be used just for the simple true or false checks.

To explain the ternary operator, let's first take a look at the syntax of a typical if statement:

```
if (condition) {  
    // value if true;  
} else {  
    // value if false;  
}
```

Now, the ternary operator:

```
condition ? value if true : value if false
```

# Ternary Operator

Although this is just a pseudocode, meaning the code written in half english and half real syntax, I think you can still see how we would use the ternary operator.

Let's use same old driver's licence example we had when we were learning about the if statement.

```
if (person.age > 18) {  
    console.log('You can drive');  
} else {  
    console.log('You may not drive yet');  
}
```

And now let's transfer it to a ternary.

# Ternary Operator

```
person.age > 18  
? console.log('You can drive')  
: console.log('You may not drive yet');
```

Reading from left to right, we first have our condition. Following a question mark is the expression that is going to be executed if the condition evaluates to true. And finally, following the colon sign is the expression that is going to be executed if the condition evaluates to false.

At the beginning, ternary operators may seem a bit weird and hard to read. But as you write more of them, you'll quickly get better at understanding them. They'll quickly become your go-to tool if you have just a simple true or false question.

# for and while loops

Sometimes we want to repeat an action a number of times for example let's imagine we want to display numbers from zero to nine on the console.

You may be thinking doing something like that:

```
console.log(0);
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
console.log(6);
console.log(7);
console.log(8);
console.log(9);
```

But that's not a good idea at all. So instead we use for or a while loop.

# for loop

The for loop is more complex, but it's also the most commonly used loop.

It's called a for loop because it runs "for" specific number of times. For loops are declared with three optional expressions separated by semicolons: initialization, condition and final-expression. Followed by a statement (usually a block statement).

```
for ([initialization]; [condition]; [final - expression]) {  
    statement;  
}
```

The initialization statement is executed one time only before the loop starts. It is typically used to define and setup your loop variable.

# for loop

The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When condition is false at the start of the iteration, the loop will stop executing. This means if condition starts as false, your loop will never execute.

The final-expression is executed at the end of each loop iteration, prior to the next condition check and is usually used to increment or decrement your loop counter.

So we already know how to use for loop let's use it to print numbers from zero to nine:

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

# for loop

First, we initialize our variable `i=0` because we start to count from 0, “`i`” stand for index and it’s kind of standard for loop variable.

Next, we set our condition to `i<10`. So every time before the loop execute the statement, it will check if the condition is true or in our example if variable ‘`i`’ is less than ten.

If it’s equal or greater than 10 then the condition will evaluate to false and terminate our loop.

Final expression is our counter update and we set it to `i++` which is shorthand for `i=i+1`. Each iteration `i` is increased by one.

# for loop

At the beginning we said that expressions are optional that mean we can skip parts, for example we can initialize loop variable before loop like that:

```
let i = 0; // we have i already declared and assigned

for (; i < 10; i++) {
  // no need for "initialization"
  console.log(i); // 0, 1, 2
}
```

We can actually remove everything, creating an infinite loop:

```
for (;;) {
  // repeats without limits
}
```

**CHAPTER 04**

# **Functions**

# Functions Intro

In this section, we're going to talk about functions. I'm really excited to show you how functions work!

Functions are one of the most interesting & most important parts of any programming language.

So what are functions, and why should we use them? A JavaScript function is a block of code designed to perform a particular task. Remember that, a block of code designed to perform a particular task. We will often need to perform a similar task many times in our application.

Functions are the main building blocks of the program. They allow the code to be called many times without repetition.

# Functions Intro

You've already seen a function in JavaScript. Not only that you've seen it, you've used it multiple times by now.

It was the function called `console.log`.

`Console.log` has a task of printing values to the console. After finishing this section, you'll be able to create your own functions as well!

When talking about functions, you're often going to hear two terms: function declaration and function call. So let's explain each one of these.

## Function:

A function is a special value with one purpose: it represents some code in your program.

# Functions Intro

Functions are handy if you don't want to write the same code many times. "Calling" a function like `sayHi()` tells the computer to run the code inside it and then go back to where it was in the program. There are many ways to define a function in JavaScript, with slight differences in what they do.

## Defining Functions

A function declaration consists of the function keyword. I'm first going to show you an example of a simple function called square:

```
function square(number) {  
    return number * number;  
}
```

# Functions Intro

A lot of stuff is written, let's review it word by word.

**function** is the reserved JavaScript keyword for creating a function.

**square** is the name of the function, you can name it however you'd like.

Then inside of parentheses we have something known as parameters.

Parameters are values we're going to send to our function when calling it. The function square takes one parameter, called number.

Names of parameters do not matter, you can name them however you'd like.

# Functions Intro

Then, we have an opening curly brace. It represents a start of the function block.

Everything else up to the closing curly brace represents the function body. In function body we can write all of the things we learned so far about JavaScript.

We can create variables, do something with operators, add if/else statements and so on.

This example function is consisted of one statement that says to return the parameter of the function (that is, `number`) multiplied by itself.

The return is really important, every function needs to have it. It specifies the value that will be returned by the function.

# Functions Intro

And how can we retrieve values from functions?

We need to call them.

Let me explain what do I mean by that.

## Calling Functions

Defining a function does not execute it. Defining it simply names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
square(5);
```

# Functions Intro

In here, we have the function name followed by parentheses, in parentheses, we put something known as arguments.

Arguments are the values we want to fill our parameters with. For example, if we send the value of 5, the parameter called number in the function declaration is going to become number 5. Then, we multiply it by itself and return it.

25

As you can see, we get 25 when we run it.

That's the return value of the called function with an argument of 5. So how can we actually use the value from the function?

# Arrow Functions

They have only one difference from "normal" functions. Arrow functions do not create their own this value. "this" is a special JS reserved keyword, we're going to explain it later in detail.

The only thing you should know right now, is that arrow functions do not create their own "this" value. In 99% percent of the cases we're not even going to need it.

The most modern way of declaring functions is using something known as arrow functions. It looks like this:

```
const square = (number) => {  
    return number * number;  
};
```

# Arrow Functions

Arrow functions also have a shorter and more concise version. Whenever we only have one return statement inside of the function and nothing else, we can return it instantly, in one line:

```
const square = (number) => number * number;
```

Arrow Functions: Arrow functions are similar to function expressions. It's one of the features introduced in the ES6 version of JavaScript

They're concise and are often used for one-liners. Arrow functions are more limited than regular functions – for example, they have no concept of this whatsoever.

# Arrow Functions

When you write this inside of an arrow function, it uses this of the closest “regular” functions above.

This is similar to what would happen if you used an argument or a variable that only exists in the function above.

Practically, this means that people use arrow functions when they want to “see” the same this inside of them as in the code surrounding them.

# Parameters vs Arguments

If you're new to JavaScript, you may have heard the terms parameters and arguments used interchangeably.

While very similar, there is an important distinction to make between these two keywords.

**Parameters** are used when defining a function, they are the names created in the function definition. Parameter is like a variable that is only meaningful inside of this function. It won't be accessible outside of the function.

**Arguments** are real values passed to the function when making a function call.

# Parameters vs Arguments

If we go back to our example, we would say that our function accepts one parameter. Parameters can be named anything.

The only thing that matters is the order. Let's try replacing name, with firstName.

```
const sayHi = (firstName) => {
  console.log(`Hi, ${firstName}`);
};

sayHi('Joe'); // Hi Joe
```

Nothing changed, our function still works. That means that parameters are just names we create for the arguments we're planning to pass into the function. As you can see, when calling the function, we have one argument.

# Parameters vs Arguments

The argument is a real JavaScript value, in this case a string of 'Joe'. Let's try adding another parameter to practice a bit more. Let's say that we want to take in both the name and the age of the person. To do that, we'd need to add a second parameter, separated by a coma.

Now, we also need to provide an argument to fill the value of age. We can do that when making a function call:

```
const logAge = (name, age) => {
  console.log(` ${firstName} is ${age} years old.`);
};

logAge('Joe', 25); // Joe is 25 years old.
```

**CHAPTER 05**

# **Tricky Concepts**

# Tricky Concepts intro

In the upcoming 3 sections we're going to cover some tricky concepts in JavaScript. **Scope**, **Hoisting** and **Closures**.

First we're going to talk about the concept of Scope, a fundamental topic for any programming language. Then we're going to mention the concepts of hoisting and closures.

I would dare to say that they are not all that useful in everyday coding. If you stucked to the good programming habits, you would never even encounter the use of hoisting.

I've still decided to include the so called "tricky concepts" because questions regarding closures and hoisting are often asked in interview questions. I've got you covered! ;)

# Scope

## What Is Scope & Why Do We Need It?

What is Scope? Why do we need it? And how can it help us write less error-prone code?

Scope simply allows us to know where we have access to our variables. It shows us the accessibility of variables, functions, and objects in some particular part of the code.

Why would we want to limit the visibility of variables and not have everything available everywhere in our code?

Firstly, it provides us with some level of security to our code. Secondly, it helps to improve efficiency, track bugs and reduce them. It also solves the problem of naming variables.

# Scope

We have three types of scopes:

1. Global Scope

2. Local Scope

3. Block Scope (only with let and const)

Variables defined inside a function are in local scope while variables defined outside of a function are in the global scope. Each function when invoked creates a new scope.

There are rules about how scope works, but usually you can search for the closest { and } braces around where you define the variable. That “block” of code is its scope.

# Global Scope

When you start writing in a JavaScript document, you're already in the Global scope.

```
const name = 'Adrian';
```

Variables written inside the Global scope can be accessed by and altered in any other scope.

```
const logName = () => {
  console.log(name);
};

logName();
```

## Advantages Of Using Global Variables

- You can access the global variable from all the functions or modules in a program

# Global Scope

- It is ideally used for storing "constants" as it helps you keep the consistency.
- A Global variable is useful when multiple functions are accessing the same data.

## Disadvantages Of Using Global Variables

- Too many variables declared as global, then they remain in the memory till program execution is completed. This can cause of Out of Memory issue.
- Data can be modified by any function. Any statement written in the program can change the value of the global variable. This may give unpredictable results in multi-tasking environments
- If global variables are discontinued due to code refactoring, you will need to change all the modules where they are called.

# Local Scope

Variables defined inside a function are in the local scope.

```
// Global Scope

const someFunction = () => {
    // Local Scope #1

    const anotherFunction = () => {
        // Local Scope #2
    };
};
```

## Advantages Of Using Local Variables

- The use of local variables offer a guarantee that the values of variables will remain intact while the task is running
- They are deleted as soon as any function is over and release the memory space which it occupies.

# Local Scope

- You can give local variables the same name in different functions because they are only recognized by the function they are declared in.

## Disadvantages Of Using Local Variables

- They have a very limited scope.

This isn't necessarily a disadvantage, but if you ever find yourself needing to use that variable in a parent scope, just declare it there. Let's me use the above example to show you what I mean.

If you need to use a variable only inside the `anotherFunction` function, just declare it there.

# Local Scope

```
// Global Scope

const someFunction = () => {
    // Local Scope #1

    const anotherFunction = () => {
        // Local Scope #2
    };
};
```

If for some reason, you need to use it both in the **someFunction** and **anotherFunction** functions, declare it in the **someFunction**.

And if you need to use it everywhere across the file, declare it in the global scope.

# Block Scope

Block statements like `if` or `for` and `while` loops, unlike functions, don't create a new scope.

Variables defined inside of a block statement will remain in the scope they were already in.

```
if (true) {  
  // this 'if' conditional block doesn't create a new scope  
  var name = 'Adrian'; // name is still in the global scope  
}  
  
console.log(name); // logs 'Adrian'
```

That is only true with the `var`.

Variables defined with `const` or `let` have something called Block scope. That means that they will be available only inside of the block of code you create them in.

# Block Scope

```
if (true) {  
    // this 'if' conditional block doesn't create a scope  
  
    // name is in the global scope because of the 'var' keyword  
    var name = 'Adrian';  
    // likes is in the local scope because of the 'let' keyword  
    let likes = 'Coding';  
    // skills is in the local scope because of the 'const' keyword  
    const skills = 'JavaScript and PHP';  
}  
  
console.log(name); // logs 'Adrian'  
console.log(likes); // Uncaught ReferenceError: likes is not defined  
console.log.skills); // Uncaught ReferenceError: skills is not defined
```

If a variable or other expression is not "in the current scope," then it is unavailable for use.

## What Is More Useful?

The local and global variables are equally important while writing a program in any programming language.

# Block Scope

However, a large number of the global variable may occupy a huge memory.

An undesirable change to global variables is become tough to identify. Therefore, it is advisable to avoid declaring unwanted global variables. Always declare variables in the scope that you want to use them in.

## KEY DIFFERENCE

- Local variable is declared inside a function whereas Global variable is declared outside the function.
- Local variables are stored on the stack whereas the Global variable are stored on a fixed location decided by the compiler.

# Local Scope

- Local variable doesn't provide data sharing whereas Global variable provides data sharing.
- Local variables are created when the function has started execution and is lost when the function terminates, on the other hand, Global variable is created as execution starts and is lost when the program ends.
- Parameters passing is required for local variables whereas it is not necessary for a global variable

# Hoisting

## What Is Hoisting?

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

This means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

Basically, when Javascript compiles all of your code, all variable declarations using var are hoisted/lifted to the top of their functional (if declared inside a function) or to the top of their global scope (if declared outside of a function) regardless of where the actual declaration has been made. This is what we mean by “hoisting”.

# Variable Hoisting

In JavaScript, an undeclared variable is assigned the value undefined at execution and is also of type undefined.

```
console.log(typeof name); // undefined
```

In JavaScript, a ReferenceError is thrown when trying to access a previously undeclared variable.

```
console.log(name);
//ReferenceError: name is not defined
```

# Variable Hoisting

Key thing to note in regards to hoisting is that the only thing that gets moved to the top is the variable declaration, not the actual value given to the variable.

```
console.log(myString); // undefined  
var myString = 'test';
```

```
var myString;  
  
console.log(myString); // undefined  
  
myString = 'test';
```

# Variable Hoisting

## Example 1:

```
var hoist;  
  
console.log(hoist); // undefined  
  
hoist = 'The variable has been hoisted';
```

## Example 2:

```
function hoist() {  
  var message;  
  
  console.log(message);  
  
  message = 'Hoisting is cool!';  
}  
  
hoist(); // undefined
```

# Variable Hoisting

## Only Declarations Are Hoisted

JavaScript only hoists declarations, not initializations. If a variable is declared & initialized after using it, the value will be undefined.

For example:

```
console.log(num); // undefined  
  
var num;  
  
num = 6;
```

To avoid this pitfall, we would make sure to declare and initialise the variable before we use it:

# Variable Hoisting

```
function hoist() {  
    var message = 'Hoisting is cool!';  
  
    return message;  
}  
  
hoist(); // Hoisting is cool!
```

The variable declaration, `var message` whose scope is the function `hoist()`, is hoisted to the top of the function.

This section of the eBook is the only time that you'll see me use the older syntax like function declarations and `var` keyword. Why is that?

It shows you that newer versions of JS are trying to get away from this way of writing code.

# Variable Hoisting

It's good to know that hoisting exists, but you should never actually use it. Always declare variables exactly where they should be: at the top of the scope they're used in. That way, your code is always going to be predictable, and you don't have to rely on hoisting.

`let` and `const` hoist but you cannot access them before the actual declaration is evaluated at runtime. What does this mean? Let's see it in a simple example:

```
console.log(myVarString); // undefined
var myVarString = 'var';

console.log(myLetString); // ReferenceError
let myLetString = 'let';

console.log(myConstString); // ReferenceError
const myConstString = 'const';
```

# Variable Hoisting

With let and const you get back exactly what you would expect: a reference error. And that's good. That's JavaScript's way of letting us know that we need to write clean code.

You should always declare variables before using them, it's common sense.

# Function Hoisting

The same as **var** variables, the function declarations are hoisted completely to the top.

```
hoisted(); // 'This function has been hoisted.'  
  
function hoisted() {  
  console.log('This function has been hoisted.');//  
};
```

Again, would you ever need to do this? No.  
Always declare the function before you call it.

## Function Expressions

Another great thing, is that constants & function expressions save us from doing that. Function expressions (the more modern way of writing functions, with `const` keyword), are not hoisted.

# Function Hoisting

```
functionExpression(); // ReferenceError  
  
const functionExpression = () => {  
  console.log('Will this work?');  
};
```

Hoisting, as well as closures, which we're going to see next, are complex topics. I would say that they are not all that useful in everyday coding.

# Closures

```
const outer = () => {
  const outerVar = 'Hello!';

  const inner = () => {
    const innerVar = 'Hi!';

    console.log(innerVar, outerVar);
  };

  return inner;
};

const innerFn = outer();

innerFn();
```

Normally, when you exit a function, all its variables “disappear”. This is because nothing needs them anymore. But what if you declare a function inside a function?

Then the inner function could still be called later, and read the variables of the outer function.

# Closures

In practice, this is very useful! But for this to work, the outer function's variables need to "stick around" somewhere. So in this case, JavaScript takes care of "keeping the variables alive" instead of "forgetting" them as it would usually do. This is called a "**closure**".

In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
const init = () => {
  const hobby = 'Learning JavaScript'; // hobby is a local variable created by init

  const displayHobby = () => {
    // displayHobby() is the inner function, a closure
    console.log(hobby); // using variable declared in the parent function
  };

  displayHobby();
};

init();
```

# Closures

`init()` creates a local variable called `name` and a function called `displayHobby()`.

The `displayHobby()` function is an inner function that is defined inside `init()` and is only available within the body of the `init()` function. Note that the `displayHobby()` function has no local variables of its own.

However, since inner functions have access to the variables of outer functions, `displayHobby()` can access the variable `name` declared in the parent function, `init()`.

Run the code and notice that the `alert()` statement within the `displayHobby()` function successfully displays the value of the `name` variable, which is declared in its parent function.

# Closures

Nested functions have access to variables declared in their outer scope.

```
const init = () => {
  const hobby = 'Learning JavaScript'; // hobby is a local variable created by init

  const displayHobby = () => {
    // displayHobby() is the inner function, a closure
    console.log(hobby); // using variable declared in the parent function
  };

  return displayHobby;
};

var myFunc = init();

myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above; what's different and interesting is that the `displayHobby()` inner function is returned from the outer function before being executed.

# Closures

At first glance, it may seem unintuitive that this code still works. In some programming languages, the local variables within a function exist only for the duration of that function's execution.

Once `init()` has finished executing, you might expect that the `name` variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A closure is the combination of a function and the environment within which that function was declared.

# Closures

This environment consists of any local variables that were in-scope at the time the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayHobby` created when `init` is run.

The instance of `displayHobby` maintains a reference to its lexical environment, within which the variable name exists.

For this reason, when `myFunc` is invoked, the variable name remains available for use and "Mozilla" is passed to `alert`.

**CHAPTER 06**

# **Strings in Detail**

# Strings Intro

In JavaScript, and in any programming language for that matter, we need a way to store text.

In JavaScript we use strings to store text. String is nothing more than a primitive data type.

How can we create strings in JavaScript?

There are a few ways:

```
const single = 'This is a string written inside of single quotes.';  
const double = "This is a string written inside of double quotes.";  
  
const backticks = `This is a string written inside of double quotes. `;
```

Strings created with single and double quotes are the same. We can call them simple or basic strings. They simply represent some static textual value.

# Strings Intro

Strings created with backticks on the other hand provide extended functionality. They are dynamic. They allow us to execute real JavaScript logic inside of them. Let me show you what I mean in an example:

```
const variableName = `${2 + 2}`; // 4
```

Everything that we put in between the dollar sign and curly brackets is not simply taken for granted. It is evaluated as JavaScript logic.

Therefore,  $2 + 2$  returns 4, rather than the string of ' $2 + 2$ '.

# Strings Intro

That means that we can also make function calls inside of backticks string, for example:

```
const sum = (a, b) => a + b;  
  
const total = `The sum is ${sum(2, 2)}`; // The sum is 4
```

Backtick strings have one extra feature. We can span them across multiple lines.

```
const numbers = `  
 1  
 2  
 3  
`;
```

If we tried doing this with basic single or double quote strings, we would get an error.

# Strings Intro

Let me ask you one question, how would the value of the following string look like?

```
const greeting = 'Hi, I'm John';
```

This would produce an error. In here, with a single quote after the I, we actually end the string. And JavaScript doesn't know how to evaluate the rest of the code.

One way to fix this would simply be to use different type of quotes. For example:

```
const greeting = "Hi, I'm John";
```

But this is not a solution. Imagine if we had both types of the strings in the sentence:

# Strings Intro

```
const greeting = "Hi, I'm John, "Doe".";
```

This would, again, break.

There's something called an "**escape character**", which allows us to treat special characters like normal letters. This is how we can escape the single and double quotes.

```
const greeting = 'Hi, I\'m John, \"Doe\". ';
```

But this is getting messy. We still have "cool" strings at our disposal, right? Let's use them.

```
const greeting = `Hi, I'm John, "Doe".`;
```

This way, we can write the string however we want. Great!

# String Escape Characters

\' Single quote

\\" Double quote

\\\ Backslash

\b Backspace

\f Form feed

\n New line

\r Carriage return

\t Horizontal tabulator

\v Vertical Tabulator

# String length and basic properties

One thing that we often want to know when it comes to strings is its length.

You might think that we need to do some complex stuff to come to that value. Like loop through all the characters, count them and then display the value. It's so much simpler than that!

```
const name = 'John';  
name.length // 4
```

Another thing we might want to do often is get the element at a certain position of the string. We can do it really easily as well.

# String length and basic properties

This is how we would get the first letter of a string:

```
name.[0] // j
```

... and this is how we would get the last letter:

```
name.[name.length - 1] // n
```

Let's inspect this line for a moment.

name.length is equal to 4 and 4 minus 1 is 3

name of 3 is the last letter, because we start from 0 and not from 1. For the reason that strings start from 0, we need to do it like this.

# String length and basic properties

... and in the same fashion, we can get any character in the string

```
name.[2] // h
```

Great! In this part, we learned how to get a length of the strings using the length property, as well as how to get certain characters of a string!

Now let's learn how we can change the case of the string!

# Change string case

In this video, we're going to learn how we can change the case of a string. What is the case? You have definitely heard about upperCASE and lowerCASE letters. That's it!

In JavaScript, we have only two, really simple and straight forward methods for changing the character case, and they are:

**string.toLowerCase()**

**string.toUpperCase()**

Let's show this in an example:

```
const string= 'Hello! How are you, James?';  
  
string.toLowerCase(); // "hello! how are you, james?"  
string.toUpperCase(); // "HELLO! HOW ARE YOU, JAMES?"
```

# Change string case

Notice how we have parentheses on these two.

That's because they are functions, more precisely: methods we call on a string.

Let's learn more useful string methods!

# Searching for a Substring

There are multiple ways to look for a substring within a string.

## str.indexOf()

The first method is `str.indexOf(substr, pos)`.

It looks for the `substr` in `str`, starting from the given position `pos`, and returns the position where the match was found or `-1` if nothing can be found.

For instance:

```
const string = 'I love ducks, he said, ducks are great!';

string.indexOf('ducks'); // 7
string.indexOf('Ducks'); // -1
```

# Searching for a Substring

The optional second parameter allows us to search starting from the given position.

For instance, the first occurrence of 'ducks' is at position 7. To look for the next occurrence, let's start the search frome position 8:

```
string.indexOf('ducks', 8); // 23
```

## str.lastIndexOf()

### str.lastIndexOf(substr, position)

There is also a similar method

**str.lastIndexOf(substr, position)** that searches from the end of a string to its beginning.

# Searching for a Substring

Great! Now you can use `indexOf` methods if you need to find the exact position of some substrings in a string.

## `includes()`

But much more often, you're just interested if a string contains something, and you're not concerned where is it in the string.

For these cases you can use `String.includes()`

It simply returns true or false.

It's the right choice if we need to test for the match, but don't need its position:

# Searching for a Substring

```
const string = 'I love ducks, he said, ducks are great!';  
  
string.includes('ducks'); // true
```

As with the `indexOf` method, the optional second argument of `str.includes` is the position to start searching from.

## `str.startsWith()` and `str.endsWith()`

The methods `str.startsWith` and `str.endsWith` do exactly what they say:

```
string.startsWith('I'); // true  
string.endsWith('ducks'); // false
```

# Searching for a Substring

The best method for getting a substring of a string is **str.slice()**.

Let's show it on an example:

**str.slice(start [, end])**

Returns the part of the string from start to (but not including) end. For instance:

```
const exampleString = 'hotdog';

exampleString.slice(3, 6); // dog
```

# Searching for a Substring

Some times, we might want to split the string into multiple substrings. For that we'll be using a string method called **split()**.

Let me give you some examples:

This is how we can split a word into characters:

```
const exampleString = 'dog';  
  
console.log(exampleString.split(''));
```

Notice how we passed an empty string as a first parameter of a split method.

# Searching for a Substring

This is how we can split a sentence into words:

```
const string = 'The quick brown fox jumps over the lazy dog.';  
console.log(string.split(' '));
```

The result of both examples is an... array!

Exactly.

# Reverse a string

## REVERSE

There isn't a built in string method that reverses a string. Rather, we can use the knowledge we previously gained! Remember how we can split a string into array of characters? Arrays do have a reverse method.

So this is a process.

1. Split a string

2. Reverse newly created a array

3. Turn the array back into string using `join()`

```
const str = 'test';  
  
str.split(' ').reverse().join(''); // tset
```

# Repeat and trim a string

## REPEAT

Let's say that you want to repeat a string an x number of times. You can easily do that by using the `string.repeat()` method.

```
const dogSays = 'woof';

console.log(dogSays.repeat(4)); // woofwoofwoofwoof
```

## TRIM

Sometimes, users don't know how to type. And we need to clean their emails, usernames & whatnot. We can clean empty spaces using trim

```
const str = "      Hello World!      ";

console.log(str.trim()); // "Hello World"
```

# Array Methods

## **string[index]**

get a certain character of a string

## **string.length**

return the number of characters in a string

## **string.split(' ')**

returns an array of words of a string

## **string.split('')**

returns an array of characters of a string

## **string.toLowerCase()**

returns a lowercased string

## **string.toUpperCase()**

returns an uppercased string

# Array Methods

## **string.charAt(index)**

returns a new string consisting of the single character located at the specified offset into the string.

## **string.replace(substr, newSubstr)**

returns a new string with a substring (substr) replaced by a new one (newSubstr).

## **string.includes(searchString)**

performs a case-sensitive search to determine whether one string may be found within another string, returns true or false.

## **string.substr(start, length)**

returns a portion of the string, starting at the specified index and extending for a given number.

# Array Methods

## **string.includes('substring')**

checks whether a substring exists inside of a string  
[check the character case]

## **string.indexOf(searchValue)**

returns the index of the first occurrence of the specified value, starting the search at fromIndex.  
Returns -1 if the value is not found.

## **string.lastIndexOf(searchValue)**

returns the index of the last occurrence of the specified value, searching backwards from fromIndex. Returns -1 if the value is not found.

## **string.slice(beginIndex, endIndex)**

extracts a section of a string and returns it as a new string, without modifying the original string.

**CHAPTER 07**

# **Arrays in Detail**

# Arrays

In programming, quite often we will need an ordered collection, where we have a 1st, 2nd, 3rd element and so on.

For example, we need that to store a list of something: users, items, elements etc.

There exists a special data structure named Array, to store ordered collections.

## Declaration

This is how we declare an array – the most important part here are the square brackets:

```
const months = [ 'January', 'February', 'March', 'April' ];
```

Array elements are numbered, starting with zero.

# Arrays

We can get an element by its number in square brackets:

```
console.log(months[0]); // 'January'
```

We can replace an element:

```
months[2] = 'Not March'; // [ 'January', 'February', 'Not March', 'April' ]
```

...Or add a new one to the array:

```
months[4] = 'May'; // [ 'January', 'February', 'March', 'April', 'May' ]
```

The total count of the elements in the array is its length:

```
console.log(months.length); // 5
```

# Arrays

An array can store elements of any type:

```
const arr = [
  'Apple',
  { name: 'John' },
  true,
  function () {
    alert('hello');
  },
];
```

You're often going to find yourself needing to loop through all the elements of an array. That's where the for loop we've learned comes in handy:

```
for (let i = 0; i < months.length; i++) {
  console.log(carr[i]);
}
```

# Arrays

Later, I've dedicated a few whole lectures about different built in array methods for looping. They allow us to loop faster, with added functionality and less code.

# Array Methods

Aside from containing variables at indexes, an array contains a variety of premade functions with which you can manipulate it's data, like adding or removing elements at certain positions! Let's take a look at a few of the most basic ones right now.

## array.push(value)

The **array.push( )** function adds a new element, containing the entered variable, to the end of the array.

```
const names = ['Jon', 'Bob', 'David', 'Mark'];  
  
names.push('Dean');  
  
console.log(names); // ["Jon", "Bob", "David", "Mark", "Dean"]
```

# Array Methods

There's one important thing that many experienced web developers don't know about `array.push()`. What's it's the return value?

Many would think that the return value of the `push` would be an array, now including the element we've pushed. Let's test it out:

```
console.log(names.push('Dean')); // 5
```

The return value is 5. Hmm, why 5? Think about it.

It turns out that the `array.push()` returns the length of the array when the element is pushed.

We can even store it in a variable:

# Array Methods

```
const length = names.push('Dean');

console.log(length); // 5
```

## array.pop()

The **array.pop()** function on the other hand, does quite the opposite, deleting the last element of an array.

```
const names = ['Jon', 'Bob', 'David', 'Mark'];

names.pop();

console.log(names); // ["Jon", "Bob", "David"]
```

This time, the return of the method is not the final length of the array, as it is with `array.push()`, rather, it's the value of the removed element. You have it in case you need it somewhere.

# Array Methods

```
const removedValue = names.pop();  
  
console.log(removedValue); // "Mark"
```

This can be great for transferring data between two arrays or just giving a value one final use before popping it into the void.

## array.shift()

Shift works almost exactly like pop, with one major difference. It deletes the first value in an array and moves the rest backwards!

```
const names = ['Jon', 'Bob', 'David', 'Mark'];  
  
names.shift();  
  
console.log(names); // ["Bob", "David", "Mark"]
```

# Array Methods

It too returns the "popped" value.

```
const removedValue = names.shift();  
  
console.log(removedValue); // "Jon"
```

## array.unshift(value)

If shift is the sister function to pop, unshift is that to push. It adds a new value to the start of an array instead of the end!

```
const names = ['Jon', 'Bob', 'David', 'Mark'];  
  
names.unshift('Dean');  
  
console.log(names); // ["Dean", "Jon", "Bob", "David", "Mark"]
```

Much like push, it returns the new array length.

# Array Methods

It too returns the "popped" value.

```
const length = names.unshift('Dean');

console.log(length); // 5
```

## array.splice()

Now this one is a little more sophisticated, but don't worry we'll walk you through it.

The splice method allows you to "splice" values into the array. It's first parameter determines where the new element or elements are placed, the second how many after that point should be deleted before placement and each subsequent condition is merely an element you wish to add.

# Array Methods

Here's an example.

```
const names = ['Jon', 'Bob', 'David', 'Mark'];

names.splice(2, 0, 'Jenny', 'Johnny');

console.log(names); // ["Jon", "Bob", "Jenny", "Johnny", "David", "Mark"]
```

It can also return an array of any deleted items, like pop!

```
const removedValue = names.splice(0, 1);

console.log(removedValue); // ["Jon"]
```

# Array Methods

## array.slice()

And finally, the slice function. This handy little piece of code can make a new variable that contains every element from a certain point on in whatever array you feed it!

```
const names = ['Jon', 'Bob', 'David', 'Mark'];

const noOneLikesJon = names.slice(1);

console.log(noOneLikesJon); // ["Bob", "David", "Mark"]
```

Don't worry, Jon is still in the first array.

```
console.log(names); // ["Jon", "Bob", "David", "Mark"];
```

# Array Methods

## **array[index]**

returns a certain value from an array

## **push(value)**

adds the value to the end of the array

## **pop()**

removes the value from the end of the array

## **shift()**

removes the value from the start of the array

## **unshift(value)**

adds the value to the start of the array

## **splice(fromIndex, no\_of\_elements)**

removes the number\_of\_elements, starting from fromIndex from the array

# Array Methods

## **slice(fromIndex, toIndex)**

copies a certain part of the array

## **concat()**

Join several arrays into one

## **join('')**

returns a string of array values

## **array.length**

returns the number of elements in the array

## **reverse()**

reverse the order of the elements in an array

## **toString()**

returns a string representing the specified array and its elements.

# Array Methods

## **toString()**

returns a string representing the specified array and its elements.

## **includes(searchElement)**

determines whether an array includes a certain value among its entries, returning true or false as appropriate.

## **sort()**

It sorts the elements of an array in place and returns the sorted array. It sorts an array alphabetically.

## **indexOf(searchElement)**

returns the index of the first occurrence of that value

## **lastIndexOf(searchElement)**

returns the index of the last occurrence of that value

# Array Methods

## array.slice()

And finally, the slice function. This handy little piece of code can make a new variable that contains every element from a certain point on in whatever array you feed it!

```
const names = ['Jon', 'Bob', 'David', 'Mark'];

const noOneLikesJon = names.slice(1);

console.log(noOneLikesJon); // ["Bob", "David", "Mark"]
```

Don't worry, Jon is still in the first array.

```
console.log(names); // ["Jon", "Bob", "David", "Mark"];
```

# Array method for looping

## array.forEach()

It executes a provided function once for each array element.

```
array.forEach( (element, index) => {  
    // code block to be executed  
})
```

## array.map()

It creates a new array populated with the results of calling a provided function on every element in the calling array.

```
array.map( (element, index) => {  
    // code block to be executed  
})
```

# Array method for looping

## array.filter()

It creates a new array with all elements that pass the test implemented by the provided function.

```
array.filter((element, index) => {  
    // code block to be executed  
})
```

## array.findIndex()

It returns the index of the first element in the array that satisfies the provided testing function

```
array.findIndex((el, idx, arr) => {  
    // code block to be executed  
})
```

# Array method for looping

## array.some()

It tests whether at least one element in the array passes the test implemented by the provided function

```
array.some((el, index, array)) => {  
    // code block to be executed  
}
```

## array.every()

It tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
array.every(element, index) => {  
    // code block to be executed  
}
```

# Array method for looping

## array.reduce()

It runs a function on each array element to produce (reduce it to) a single value. It works from left-to-right.

```
array.reduce((prevValue, currentValue,  
currentIndex, array)) => {  
    // code block to be executed  
}
```

## array.reduceRight()

It runs a function on each array element to produce (reduce it to) a single value. It works from right-to-left.

```
array.reduceRight((accumulator,  
currentValue, index, array)) => {  
    // code block to be executed  
}
```

**CHAPTER 08**

# **Objects in Detail**

# Intro to Objects

Objects are the most important data type and building block for modern JavaScript.

Objects are quite different from JavaScript's primitive data-types (numbers, strings, booleans...) in the sense that while these primitive data-types all store a single value, objects can store multiple.

Objects in JavaScript can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

In JavaScript, an object is a standalone entity, with properties and type.

# Intro to Objects

Compare it with a cup, for example. A cup is an object, with properties. A cup has a cooler, a design, weight, a material it is made of, etc.

The same way, JavaScript objects can have properties, which define their characteristics.

**So what are objects, why do we need them and how can we declare them?**

In simple words, object is an unordered collection of related data in form of key and value pairs.

# Intro to Objects

Let's create a simple object so that we can see everything in action:

```
const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 25,  
};
```

In here, we can see that to create an object, we need to just open a set of curly brackets & assign it to a variable, that's the form of the object.

Then, inside, we have key and value pairs. For instance, first key in the person object is firstName, with a corresponding value of 'John'. Inside of there, we also have lastName and age.

# Intro to Objects

And all of these properties are somehow grouped together. Each of these keys is referred to as properties of the object.

Opposed to having something like this, where every value stands for itself, objects allow us to make certain pieces of data related.

```
const firstName = 'John';
const lastName = 'Doe';
const age = 25;
```

Also, we mentioned that objects are unordered, what does that mean? The order of properties in an object can change, and will not always stay the same as we declared it. But that's fully okay.

# Intro to Objects

But that's fully okay. When we're working with objects, we are not worried about the order, that's what arrays are for, more on that really soon! :)

In our person object, you can see that our firstName & lastName are strings, & age is an object. Values in an object can be of absolutely any type. For example, let's say that our person has a car, that car, can again be an object, because there are a lot of properties we want to describe that car with:

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 25,
  car: {
    year: 2015,
    color: 'Red',
  },
};
```

# Intro to Objects

That simply means that we can nest objects inside other objects.

We can also add variables as values in an object. In a sense, a variable is just a box that holds a value. Because of that, we can do the following:

```
const firstName = 'Johnny';

const person = {
  firstName: firstName,
};

// show the output
```

... one thing that we can notice is that the key and value have the same name. If that is the case, JavaScript allows us to write it like this:

# Intro to Objects

```
const person = {  
  firstName,  
};  
  
// show the output
```

meaning, just remove one, and it is still going to work.

# Accessing, adding and updating properties

## Dot Notation

To access, add, or update a property of an object, we use something called dot notation.

Dot notation allows us to retrieve some values from an object, for example, if we want to get only the `firstName` of our person.

we can do the following:

```
person.firstName;
```

In the same fashion, we can add or overwrite some properties:

```
person.dog = { name: 'Fluffy', age: 2 };
```

# Accessing, adding and updating properties

```
person.age = 26;
```

## Square Bracket Notation

There is also a second way to retrieve properties from an object, it is called **square bracket notation**.

Square bracket notation, like the dot notation, also allows us to access properties of an object:

```
person['firstName'];
```

...but it also has some additional features, for example, we can access properties dynamically. Let me show you what I mean:

# Accessing, adding and updating properties

```
const property = 'age';  
  
person[property];
```

It is also used when we have key names that are not usual JavaScript variable names:

```
// add 'this-is-a-key-with-dashes'  
// and 'this is another key' to the person object.  
  
person['this-is-a-key-with-dashes'];  
person['this is another key'];  
  
// person.this is another key - error  
// person.this-is-a-key-with-dashes - error
```

# Object Methods

A method is a function associated with an object, or, simply put, a method is a property of an object that is a function.

Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object.

```
objectName.methodname = functionName;

var myObj = {
  myMethod: function(params) {
    // ...do something
  }

  // OR THIS WORKS TOO

  myOtherMethod(params) {
    // ...do something else
  }
};
```

# Object Methods

Where `objectName` is an existing object, `methodname` is the name you are assigning to the method, and `functionName` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname( params );
```

You can define methods for an object type by including a method definition in the object constructor function. You could define a function that would format and display the properties of the previously-defined Car objects; for example,

# Object Methods

```
function displayCar() {  
  var result = `A Beautiful ${this.year} ${this.make} ${this.model}`;  
  pretty_print(result);  
}
```

where `pretty_print` is a function to display a horizontal rule and a string.

Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `Car` by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of `Car` would now look like

# Object Methods

```
function Car(make, model, year, owner) {  
    this.make = make;  
    this.model = model;  
    this.year = year;  
    this.owner = owner;  
    this.displayCar = displayCar;  
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar;  
car2.displayCar;
```

**CHAPTER 09**

# **Value vs Reference Intro**

# Value vs Reference

JavaScript differentiates Data Types on:

- **Primitive values**

(Number, String, Boolean, Null, Undefined...)

- **Complex values**

(Objects, Arrays)

## Copying Primitive Values:

When copying primitive values, JavaScript is going to behave as we expect it to. You just need to see what was the value of the variable at the time of the assignment.

Let me give you a few real examples:

# Value vs Reference

## Copying Numbers

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

x = 2;

console.log(x); // 2
console.log(y); // 1
```

## Copying Strings

```
let firstPerson = 'Mark';
let secondPerson = firstPerson;

firstPerson = 'Austin';

console.log(firstPerson); // Austin
console.log(secondPerson); // Mark
```

# Value vs Reference

## Copying Complex Values:

When copying complex values, JavaScript engine is not going to behave as you would initially think it would.

Let me give you an example:

## Copying Arrays

```
const animals = ['dogs', 'cats'];
const otherAnimals = animals;

animals.push('llamas');

console.log(animals); // [ 'dogs', 'cats', 'llamas' ]
console.log(otherAnimals); // [ 'dogs', 'cats', 'llamas' ]
```

★ Complex Values are copied by reference

# Value vs Reference

Wait what?! What happened here? Why are both arrays the same if we only pushed the value to the first array?

Something just doesn't feel right, right?

Let's try something similar with objects to see whether the behaviour continues.

## Copying Objects

```
const person = {  
  firstName: 'Jon',  
  lastName: 'Snow',  
};  
  
const otherPerson = person;  
  
person.firstName('JOHNNY');  
  
console.log(person); // { firstName: 'JOHNNY', lastName: 'Snow' }  
console.log(otherPerson); // { firstName: 'JOHNNY', lastName: 'Snow' }
```

★ Primitive Values are copied by reference

# Value vs Reference

What?! Again, this doesn't look right. So... what happened here? Well...

When a variable is assigned a primitive value, it just copies that value. We saw that with number and strings examples.

On the other hand, when a variable is assigned a non-primitive value (such as an object, an array or a function), it is given a reference to that object's location in memory.

What does that mean?

In this example above, the variable `otherPerson` doesn't actually contain the value `{ firstName: 'Jon', lastName: 'Snow' }`, instead it points to a location in memory where that value is stored.

# Value vs Reference

What?! Again, this doesn't look right. So... what happened here? Well...

```
const otherPerson = person;
```

When a reference type value is copied to another variable, like otherPerson in the example above, the object is copied by reference instead of value. In simple terms, person & otherPerson don't have their own copy of the value. They point to the same location in memory.

```
person.firstName('JOHNNY');

console.log(person); // { firstName: 'JOHNNY', lastName: 'Snow' }
console.log(otherPerson); // { firstName: 'JOHNNY', lastName: 'Snow' }
```

# Value vs Reference

When a new item is pushed to `person`, the array in memory is modified, and as a result the variable `otherPerson` also reflects that change.

We're never actually making a copy of a `person` object. We're just make a variable that points to the same location in the memory.

## Equality

We can prove that with a simply equality check.

```
const person = { firstName: 'Jon' };
const otherPerson = { firstName: 'Jon' };

console.log(person === otherPerson);
```

What do you think? Are `person` and `otherPerson` equal? Well, they should be, right?

# Value vs Reference

They look exactly the same, have the same keys and values. Let's check it out:

```
console.log(person === otherPerson); // false
```

You might expect `person === otherPerson` to resolve to true but that isn't the case. The reason behind that is that although `person` and `otherPerson` contain identical objects, they still point to two distinct objects stored in different locations in memory.

Now, let's create a copy of the `person` object by copying the object itself, rather than creating a completely new instance of it.

# Value vs Reference

```
const anotherPerson = person;  
  
console.log(person === anotherPerson); // TRUE
```

`person` and `anotherPerson` hold reference to the same location in memory & are therefore considered to be equal.

Awesome! We just learned that primitive values are copied by value, and that objects are copied by reference.

Up next we're going to learn how to make a real copy of an object. That will allow us to copy an object and change it without being afraid that we'll change both objects at the same time.

# Shallow Cloning

We've seen all the problems we could possibly encounter if we tried changing values of an object copied by a reference. So how can we properly copy it and remove a reference? How can we create a complete clone of the object?

## Cloning Arrays:

### 1st way: Spread Operator

Spread operator is a newer addition to JavaScript. Before using it to clone an object without keeping a reference.

Let's first explore how it works.

Imagine you had an array:

# Shallow Cloning

```
const numbers = [1, 2, 3, 4, 5];
const newNumbers = [...numbers];
```

★ To clone an array, use the spread operator

How could we use the spread operator on this array? Spread syntax allows us to "spread" the values of strings, objects and arrays.

How does it work? Let's see it in action on our numbers array.

The syntax of a spread operator is represented just by three dots.

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

# Shallow Cloning

We get back all the values from the array individually, one after the other, they got taken out of the array. So how can we make use of this?

Take a look at the following code. In there, we created a new variable, and in it we put a completely new, empty array, in which we spread the values of our original array.

```
const newNumbers = [...numbers];
```

Let's do a simple `console.log`:

```
console.log(newNumbers); // [1, 2, 3, 4, 5]
```

# Shallow Cloning

We get an array that looks identical to the one that we had at the beginning, but is it completely identical? Let's check the equality. We're going to create another copy and then compare them.

```
const copiedNumbers = numbers;  
  
console.log(numbers === copiedNumbers); // true  
console.log(numbers === newNumbers); // false
```

We got back true and false. What does this mean? This means that the `copiedNumbers` array is pointing to the same place in memory where the original `numbers` array is pointing to.

Therefore, if we change one, or the other, they would both change. And we do not want that.

# Shallow Cloning

On the other hand, numbers and newNumbers are not the same. They represent a completely different array. Let's try changing the original numbers array:

```
numbers.push(6);

console.log(numbers); // [ 1, 2, 3, 4, 5, 6 ]
console.log(copiedNumbers); // [ 1, 2, 3, 4, 5, 6 ]
console.log(newNumbers); // [ 1, 2, 3, 4, 5 ]
```

As you can see, the numbers & copiedNumbers both got changed. And the array with we created using the spread operator was unchanged! This means that we created something called a "shallow clone". The "shallow" part is going to make sense once we introduce the "deep clones".

# Shallow Cloning

## 2nd way: Array.slice()

```
numbers = [1, 2, 3, 4, 5];
numbersCopy = numbers.slice();
// [1, 2, 3, 4, 5]
```

# Cloning Objects

## 1st way: Spread Operator

We can clone the objects using the spread operator as well.

```
const person = {
  name: 'Jon',
  age: 20,
};

const otherPerson = { ...person };
```

# Shallow Cloning

... now we can change the `otherPerson` without changing person:

```
otherPerson.age = 21;  
  
console.log(person); // unchanged  
console.log(otherPerson); // changed
```

## 2nd way: `Object.assign()`

```
var A1 = { a: '2' };  
var A2 = Object.assign({}, A1);
```

Awesome! We just learned two different ways for cloning both objects and arrays! As I mentioned, the ways of cloning we just explored create SHALLOW CLONES.

# Deep Cloning

```
const person = {  
  firstName: 'Emma',  
  car: {  
    brand: 'BMW',  
    color: 'blue',  
    wheels: 4,  
  },  
};
```

Now.. Let's try creating a copy of that object, in one of the ways we've learned now:

For example, we can use the spread operator:

```
const newPerson = { ...person };
```

Great, we've learned that this removes the reference from the original object, right?

So let's try changing the newly created object:

# Deep Cloning

```
newPerson.firstName = 'Mia';
```

Let's console logging both objects and see what changed? Without reading you can try to answer.

```
console.log(person); // unchanged  
console.log(newPerson); // changed
```

As we learned, if we use the spread operator, the reference to the initial object gets deleted, therefore, we can change the new object without having to worry!

Oh, if only it were that simple... Let's see what would happen if we tried changing the properties of Mia's car.

# Deep Cloning

```
newPerson.car.color = 'red';
```

... and then console log both objects:

```
console.log(person); // changed  
console.log(newPerson); // changed
```

Both objects got changed. How did that happen?

Well... we only remove the reference from the outer object, the person one, but notice how the car is also an object...

It has it's own reference, and the same rules apply.. If we want a real copy, we need to remove a reference from the inner object as well.

# Deep Cloning

We could do that by spreading the properties of an inner object:

```
const newPerson = { ...person, car: { ...person.car } };
```

If we now tried changing the properties of the car, it would only change them on the newly created object, let's try it out:

```
newPerson.car.color = 'red';  
console.log(person); // unchanged  
console.log(newPerson); // changed
```

That's great! But this only works for two levels into depth, first level is when we're inside of the person object, and second level is when we're inside of the car object.

# Deep Cloning

But if we had more nested objects, we'd need to spread everything. And that's not the solution.

When we have deeply nested objects, we need to create a deep clone. For an object to be a deep clone, it needs to destroy all the references.

There are two methods that are going to make this extremely easy for us. First one is called `JSON.stringify` and the second one is called `JSON.parse`. Let's see them in action, we can use the same person object we declared above:

```
const person = {
  firstName: 'Emma',
  car: {
    brand: 'BMW',
    color: 'blue',
    wheels: 4,
  },
};
```

# Deep Cloning

... first, we're going to use JSON. stringify method. The JSON.stringify() method converts a JavaScript object or any value to a string. During this process, all the references are destroyed.

```
const stringifiedPerson = JSON.stringify(person);

console.log(stringifiedPerson);
```

The thing that we get back is a string... That isn't all that valuable to us. How do we turn it back to an object again? We can do that by using JSON.parse method:

```
const newPerson = JSON.parse(stringifiedPerson);

console.log(newPerson);
```

# Deep Cloning

The `JSON.parse()` method parses a string, constructing the JavaScript value or object contained in the string.

Before we start testing it out, there's one simple tweak we can make. We can do it in one line:

```
const newPerson = JSON.parse(JSON.stringify(person));
```

Awesome! We got our object back. Let's prove that all of the references are indeed deleted and that the `newPerson` is indeed a deep clone.

```
newPerson.firstName = 'Mia';
newPerson.car.color = 'red';

console.log(person); // unchanged
console.log(newPerson); // changed
```

# Deep Cloning

That's it! You've just mastered one of the hardest topics in the whole of JavaScript! If you're still a bit confused, that's completely normal.

I'd suggest rereading this section. Take it slow. Mastery takes time.

**CHAPTER 10**

**DOM –  
Document Object Model**

# Intro to DOM

DOM (Document Object Model) is a standard to access and share documents over internet.

It represents how a particular document is structured. It helps programming languages like JavaScript and others to understand, modify the document.

HTML, XML etc have different object models. we will focus on HTML DOM

HTML DOM defines how an html page is structured, how it can be modified, properties and styles that can be added. various valid elements that can be used to structure a html page, version of the standard to be followed etc.

# Intro to DOM

HTML DOM is a tree of nested html elements that are defined while designing the html page.

Certain rules must be followed as per the standard while defining the html page

DOM is not a language but more of a standard to represent a document over the web. DOM is a set of nested tree where each node represents a particular html element.

Core DOM is defines the basic things for all documents while specific DOMs like HTML DOM extends the core DOM to support HTML documents. Similarly XML and other documents.

# Intro to DOM

## Accessing DOM

DOM elements can be accessed and modified using the methods.

A simple example is to get notified once DOM is available for use and we can proceed to manipulate it as and when needed

```
<body onload="window.alert( "document loaded" )"></body>
```

Another example is to add content to DOM dynamically once it is loaded. we will display todays date once the document is loaded

# Intro to DOM

```
<html>
  <head>
    <script>
      window.onload = function() {
        const heading = document.createElement("h1");
        const date = document.createTextNode(new Date().toString());
        heading.appendChild(date);
        document.body.appendChild(heading);
      }
    </script>
  </head>
</html>
```

In above examples we are using JavaScript language with used DOM methods like createElement or createTextNode or appendChild to modify the DOM.

Also we are using DOM events like onload to do something when that event has fired. Although there are many events, the most commonly used are click, focus, blur

# Selecting Elements

As we are aware one of the things we can do with JS is to modify or manipulate existing elements.

Consider a simple scenario that we want to display the same thing below a input what user enters in it.

We want to manipulate the element below the input somehow. The first step which we need to do is to identify or you can say select the element. we cannot modify something unless we know what to modify. We are here aware that we want to modify the element below an input. lets see how we can do the same in javascript.

There are many ways to select an element.

# Selecting Elements

**Finding HTML elements by id** – The Easiest and the most efficient way to find an element id by id. We can assign some id to an element and can select that element as shown below.

focus  
click  
blur

mostly used actions

One thing that needs to be remembered is that all elements of a single web page should have unique ids.

Consider that need for this as if there was same enrollment ids for multiple students in a college, then it would be difficult for college for further process like managing records for a particular student, his fees history, grade history etc.

# Selecting Elements

```
...  
  
<div id="element-below-input"></div>  
  
...  
  
<script>  
  
var element = document.getElementById("element-below-input");  
  
</script>
```

**Finding HTML elements by tag name** – consider that for some reasons we want to manipulate all elements of a particular kind . Lets say we want to manipulate all images of our webpage which can be done as shown below.

```
...  
  
  
  
  
  
...  
  
<script>  
  
var images = document.getElementsByTagName("img");  
  
</script>
```

# Selecting Elements

## Finding HTML elements by class name

– Class names are assigned to elements to style them uniquely. lets say to style all links in our website we had assigned some class say "our-link" to them and now we want to manipulate them.

## Finding HTML elements by CSS selectors

– css selectors are that you can use any combination of class, id, tag name and many more to select a particular element.

We will not go into detail as it itself can be another topic. For example lets just say that we had given a same class "link" to all our links in web page as well as some span elements also to make them look like a link but now we just want to select all the span elements and not the links.

# Selecting Elements

```
...  
Link1  
Link2  
Link3  
Made to look like link  
Also Made to look like link  
...  

```

**"span.link"** – "span" is used to refer all span elements and ".link" is used to refer all elements with class "link". Combining both as above we are selecting all span elements with class "link"

# Element Properties and Methods

HTML elements can have different attributes assigned to them like id, class, type etc.

There are standard attributes for different elements and when a browser identifies a standard attribute a corresponding DOM property is created and assigned to the element.

These happens only for standard attributes only. Some attributes are applicable to all elements while some are applicable to particular ones only.

For example id, class ... are applicable to all elements. While type is applicable to input element and button.

# Element Properties and Methods

if you assign type to a div and try to access it it is will be undefined.

Attributes is what we assign in html. Properties is stored in DOM objects

Some most used properties and methods are as below more can be found at

## Properties

### `classList`

a list of class assigned to an element - not a string its a array

### `id`

string of the id assigned to an element

# Element Properties and Methods

## `classname`

string of classes assigned to an element - if there are more than one class separated by space

## `innerHTML`

inner content of the element - if has nested elements then in string form

## Properties

### `addEventListener`

listen to a any type of event and call some function when that event has occurred. types can be click, mousedown, mouseup, focus, blur etc.

### `hasAttribute`

method to check if an element has an attribute or not

# **Element Properties and Methods**

## **removeAttribute**

remove some attribute assigned to an element

## **getBoundingClientRect**

returns the height, width, left and top values of an element related to the browser

## **removeEventListener**

remove event listener from an element

## **scroll**

scroll to an element position

# Element Properties and Methods

Usage of above properties is shown below

```
...
<div id="div1" class="class1 class2"><span>hello</span></div>

<input id="input" type="text"></input>

<button> Button </button>

...
<script>

var element= document.getElementById("div1");

console.log(element.classList); // ["div1", "div2", value: "div1 div2"]

console.log(element.className); // "div1 div2"

console.log(element.id); // "div1"

console.log(element.innerHTML); // "<span>hello</span>"

</script>
```

# Working with classes

Mostly developers think classes are used mostly to style the elements.

But it can also be used by javascript to do something to elements with a certain class.

Any number of class names can be given to any of the HTML element, there isn't any restriction on that. However in case there are more than one class name being assigned to a HTML element, it must be separated by space " " between them.

See below example for reference.

# Working with classes

```
<head>
  <style>
    .menu-item{
      background-color: black;
      color: white;
    }
    .menu-item.active{
      background-color: white;
      color: black;
    }
  </style>
</head>

<body>
  <ul>
    <li class="menu-item">Menu 1</li>
    <li class="menu-item active">Menu 2</li>
    <li class="menu-item">Menu 3</li>
  </ul>
</body>;
```

That was an example of how class is used to style elements.

Now lets see how to mix it up with javascript.

First we need to select the elements with certain class and for that "**getElementsByClassName( )**" method is used.

# Working with classes

We will add active class to the menu item when user clicks it and remove from the previous menu items if there any.

Lets modify our above code to do that.

```
<head>
    <script>

        <style>
            .menu-item{
                background-color: black;
                color: white;
            }
            .menu-item.active{
                background-color: white;
                color: black;
            }
        </style>
    </script>
</head>
```

```
function menuClicked(currentElement){

    const menuItems = document.getElementsByClassName("menu-item");

    for (var i = 0; i < menuItems.length; i++) {

        menuItems[i].classList.remove('active');

    }

    console.log(currentElement);
    event.target.classList.add("active");
}
```

# Working with classes

```
<body>
  <ul>
    <li class="menu-item" onclick="menuClicked(this)">
      Menu 1
    </li>

    <li class="menu-item active" onclick="menuClicked(this)">
      Menu 2
    </li>

    <li class="menu-item" onclick="menuClicked(this)">
      Menu 3
    </li>
  </ul>
</body>;
```

The above code works as we want but its not optimized. I will keep it upto you to figure what more can be done to make it better.

# Creating, Traversing and Removing Nodes

## Creating

There are different ways to create html using javascript. The mostly used way is document.createElement method.

## Syntax

```
document.createElement('tagname'); // tagname can be any valid html tag
```

There can be some more options added to it but those are related to web components which we don't need to focus right now.

But that's an interesting new thing if any one wants to peek into it.

# Creating, Traversing and Removing Nodes

The created html element can be added attributes and content using the element properties and methods that we have already seen how to use them.

The above method just creates the element but doesn't add it to the DOM. To add it to the DOM we will use appendChild method

We can add it to any element or the main body also. We will get reference to the element to which we want to add new element using query selector.

# Creating, Traversing and Removing Nodes

## Traversing Dom

Sometimes we are not sure about the element to be manipulated directly but we are sure that we know something around it either its parent or children or something. There are some methods that can be used in such case

```
<ul class="subjects">
  <li>Maths</li>

  <li class="fav-subject">Science</li>

  <li>English</li>
</ul>;
```

# Creating, Traversing and Removing Nodes

```
<script>

const subjects=document.querySelector(".subjects");

console.log(subjects.firstElementChild); // prints first element of list

console.log(subjects.lastElementChild); // prints last element of list

const favSub=document.querySelector(".fav-subject");

console.log(favSub.previousElementSibling); // prints element before favorite subject.

console.log(favSub.nextElementSibling); // prints element after favorite subject

console.log(favSub.parentElement); //prints parent of favorite subject i.e entire list

</script>
```

We have similar methods to traverse with nodes

- **ele.childNodes**
- **ele.firstChild**
- **ele.lastChild**
- **ele.previousSibling**
- **ele.nextSibling**
- **ele.parentNode**

# Creating, Traversing and Removing Nodes

## Removing Nodes

Another major thing that is needed while manipulating html is removing elements that are not needed after some action.

Lets say there was one link for login and once user has logged its not needed anymore. so we need to remove that from DOM.

```
const favSub = document.querySelector('.fav-subject');

favSub.remove(); // removes element from DOM
```

remove method only removes element from DOM but it is still present in memory and can be added again as and when needed.

## CHAPTER 11

# Classes, "new" and "this"

# The "new" Keyword

new keyword have multiple aspects related to it but for now lets just consider the most basic functionality the new keyword performs.

**It creates a new object.**

Yes that's it . Not so difficult to grasp right ? new keyword creates a new empty object . Enough of talking lets dive in to code and create same person object from our last lecture but this time using the new keyword

```
const person = new Object();
```

what this single line of code did is created an empty object called "person" . by empty I mean there are no properties attached to it , its literally empty like **person={} .**

# The "new" Keyword

We can treat this object same like the person object from our previous example .We can add new properties to this object like we did before

```
newPerson.lastName = 'John';
```

and we can access these properties exactly like before

```
console.log(person.lastname); // John
```

lets see the typeof person object created with new keyword

```
console.log(typeof(person)); // object
```

# The "new" Keyword

So the most basic thing that new keyword performs is create an empty object

You might be wondering that what in the line `const person = new Object();` what the `Object()` keyword is? and why its used in combination to the `new` keyword ?

Well don't worry lets explore this

## Object Methods, Object()

`Object()` also known as Object Constructor or Object Method is default constructor method provided by javascript to create objects.

Javascript provide us ability to create our own object constructors and create new objects from its type. Don't believe me ? lets give it a try.

# The "new" Keyword

```
function person(name, age, profession) {  
    this.name = name;  
    this.age = age;  
    this.profession = profession;  
}  
  
const john = new person('John', 23, 'Teacher');  
console.log(john.name);
```

what we did is instead of using a default constructor provided by js to us we created our own constructor method "Person" and than created an object of Person type out of it.

we can also create our object like

```
const john = new Person();
```

# The "new" Keyword

creating an object with empty constructor would initialise its properties with undefined and not null

I know the **this** keyword is bothering you but don't worry its coming next

## 'new' and the 'this' Keyword

**this** keyword is treated differently depending on the execution context but for now lets only discuss the relation between this and the new keywords

| "new keyword binds this to object itself"

# The this Keyword

So first of all what is the "This keyword" and what is it used for, well the this keyword is used to reference the object that is executing the the current function, in other words every function has a reference to it's current execution context.

For example if we want to create a function that console.log's a string using this keyword it would a little like this.

```
function Sentence(words) {  
    this.words = words;  
  
    console.log(this);  
}  
  
const S = new Sentence('learning about This keyword');
```

# The **this** Keyword

So let's see what is going on here.

First off we set a function with input "words", and inside this we set "this.words" to equal to our input, then we console.log this and create a variable that contains a new input for our sentence, but since this keyword equals our input, it'll console.log our new input.

CHAPTER 12

# Asynchronous JavaScript

# Intervals and Timers

Welcome to a new module in this eBook! I am super excited for this one, because now we're going to understand some important concepts in Javascript. These concepts will later help us provide functionalities to our app like fetching data and sending data to servers.

Until now, in this eBook we have been dealing with something called synchronous javascript and it's now time to understand asynchronous JS.

You might not understand these terms, but by the end of this lecture you will know what are they and why these concepts are crucial to building real world apps. So, let's get started!

# Intervals and Timers

In JavaScript, there is a variety of pre-made functions that allow you to execute chunks of code in timed intervals, even while other code in the program is being executed.

Imagine you're coding a video game and need a function that could execute certain drawing operations every millisecond, or a local clock that ticks the amount of time a user has spent on your site; you'd probably help yourself with the following:

## **setInterval():**

The setInterval allows you to execute a chunk of code every time a specified amount of milliseconds passes.

# Intervals and Timers

For example; this code logs "Hello World" every thousand milliseconds:

```
setInterval(() => {
  console.log('Hello World');
}, 1000);
```

That's great and all, but how do you prevent an interval from going on forever? Or store one for that matter. Well, any interval can be stored as a full variable that can later be cleared using the clearInterval() function. A more professional take on the example above would then be:

```
const myInterval = setInterval(() => console.log('Hello World'), 1000);

clearInterval(myInterval);
```

# Intervals and Timers

The clear function is especially useful if you only want an interval to execute a certain amount of time, and clear it once a condition (such as a milisecond counter getting to a certain value) is reached.

## setTimeout():

The setTimeout function allows you to wait a certain amount of time before executing a chunk of code, do note that other code outside of the timeout will continue execution as normal. The way it's used is identical to setInterval.

```
const myTimeout = setTimeout(() => console.log('Hello, World'), 1000);
// console log's "Hello World" after a thousand ms

clearTimeout(myTimeout);
```

# Intervals and Timers

It is cleared using the function `clearTimeout()`;

This is the first time you see JavaScript code that doesn't execute linearly, from top to bottom. It is asynchronous. The code on top can be executed after the code on the absolute bottom of the file. Later in the course, we're going to go into much more depth in terms of asynchronous JavaScript.

# Introduction to Asynchronous JavaScript

## Synchronous JS Example

What is synchronous Javascript?

Synchronous Javascript is one in which the code is executed line by line and their tasks are completed instantly, i.e. there is no time delay in the completion of the tasks for those lines of code.

First let me give you an example of synchronous javascript:

# Introduction to Asynchronous JavaScript

```
const functionOne = () => {
    console.log('Function One');

    functionTwo();
    console.log('Function One, Part 2');
};

const functionTwo = () => {
    console.log('Function Two');
};

functionOne();

// Function One
// Function Two
// Function One, Part 2
```

As expected, first 'Function one.' is logged and then fnTwo is invoked, so 'Function two.' is logged and then back in fnOne, 'Function one, part 2' is logged. Pretty straight and simple, isn't it?

# Introduction to Asynchronous JavaScript

## Let's change the functions

It's time to give you a taste of asynchronous javascript! So, let's change the functions we wrote:

```
const functionOne = () => {
    console.log('Function One'); // 1

    functionTwo();

    console.log('Function One, Part 2'); // 2
};

const functionTwo = () => {
    setTimeout(() => console.log('Function Two'), 2000); // 3
};

functionOne();

// Function One
// Function One, Part 2
// (after two second delay)
// Function Two
```

# Introduction to Asynchronous JavaScript

Here, in fnTwo instead of a normal console.log, we will use a setTimeout in order to be able to fake a time delay that happens when we are fetching data from servers or interacting with APIs (Application Programming Interfaces).

So, for the setTimeout we will have a callback function wherein we will log 'Function two.' to the console. We will keep 2000 millisecond delay.

So, most probably you would be surprised that the code didn't behave the way you thought it should and it's completely normal to be wrong here, right now.

# Introduction to Asynchronous JavaScript

Here, you might think, why does the javascript engine not wait for the setTimeout to end and then continue. So, let's say, we might have some asynchronous code in our script and after those lines we have some code to handle DOM events.

So if the JS engine stops for the things which take time, the users might interact with the webpage at that time and those events will remain unhandled leading the user to think that the website is not working! That is not good. So you see that, this feature is in fact in our favor, we just have to handle these kind of functions differently.

So , let's define Asynchronous Javascript

# Introduction to Asynchronous JavaScript

Asynchronous Javascript is one in which some lines of code take time to run. These tasks are run in the background while the Javascript engine keeps executing other lines of code. When the result of the asynchronous tasks gets available, it is then used in the program.

So, the main concept behind Asynchronous Javascript is that we don't wait for a function to get executed and complete its task and then handle the result.

But, we simply let the `async` function do its job in the background and we move on to execute the other lines of code and then use the result of that asynchronous task when it is available.

# Async JavaScript and Callback Hell

In this section we're going to cover a lot of advanced JavaScript concepts, some of which are: API data fetching, asynchronous code, callback functions, promises and `async/await`.

This section is going to teach you how you can first simulate or create that asynchronous source and then how we can properly deal with the data coming out of it.

Let's immediately dive in into a real example of asynchronous JavaScript, and that is: data fetching. With JavaScript, we can fetch the data from a range of different API's. API stands for Application Programming Interface & it is simply something that you can access data from

# Async JavaScript and Callback Hell

Once we fetch the data, depending on the size of the data we're fetching and our internet speed, the fetching is going to take a certain amount of time.

Opposed to the `setTimeout`, where we always waited for 2 seconds, with real data fetching, we cannot be sure how long is it going to take.

Now we're going to simulate that data fetching.

Let's say that we're trying to fetch a user from the database.

This is the problem:

# Async JavaScript and Callback Hell

```
const fetchUser = (username) => {
  setTimeout(() => {
    return { user: username };
  }, 2000); // we dont know the time, but let's say 2000
};

const user = fetchUser('test');

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

The reason because we got undefined is that the data wasn't return from the function immediately. It waited 2 seconds. How can we fix this?

Here comes the concept of Callback Functions.

We can pass in a callback function that's going to run when the data is fetched

# Async JavaScript and Callback Hell

```
const fetchUser = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the user');

    callback({ user: username });
  }, 2000);
};

const user = fetchUser('test', (user) => {
  console.log(user);
});
```

Our fetch user function accepts a callback function as a parameter, and that's where we get the data.

# Async JavaScript & Callbacks Part 2

This was just a simple example, but now, let's add more things onto it, because later on, we're going to exchange callback functions for both Promises and Async/Await .

To complicate it, we're going to imagine that we're working on a social media platform of sorts. Once the user profile is fetched, then we want to fetch his photos.

So let's create a function for that:

# Async JavaScript & Callbacks Part 2

```
const fetchUser = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the user');

    callback(username);
  }, 2000);
};

const fetchUserPhotos = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the photos');

    callback(['photo1', ['photo2']]);
  }, 2000);
};

const user = fetchUser('test', (username) => {
  console.log(username);

  fetchUserPhotos(username, (userPhotos) => {
    console.log(userPhotos);
  });
});
```

This is already getting messy.

# Async JavaScript & Callbacks Part 2

Let's add just one more function, and you'll easily notice the problem.

```
const fetchUser = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the user');

    callback(username);
  }, 2000);
};

const fetchUserPhotos = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the photos');

    callback(['photo1', 'photo2']);
  }, 2000); // we dont know the time, but let's say 2000
};

const fetchPhotoDetails = (photo, callback) => {
  setTimeout(() => {
    console.log('Now we have the photo details');

    callback('details...');
  }, 2000); // we dont know the time, but let's say 2000
};

const user = fetchUser('test', (username) => {
  console.log(username);

  fetchUserPhotos(username, (userPhotos) => {
    console.log(userPhotos);

    fetchPhotoDetails(userPhotos[0], (details) => {
      console.log(details);
    });
  });
});
```

# Async JavaScript & Callbacks Part 2

As you can see, if we use callbacks, we get this weird structure that just keeps moving to the right. If we added a few more callbacks, this is how it would look:

```
const user = fetchUser('test', (username) => {
  fetchUserPhotos(username, (userPhotos) => {
    fetchPhotoDetails(userPhotos[0], (details) => {
      fetchPhotoDetails(userPhotos[0], (details) => {
        fetchPhotoDetails(userPhotos[0], (details) => {
          fetchPhotoDetails(userPhotos[0], (details) => {
            console.log(details);
          });
        });
      });
    });
  });
});
```

This is called callback hell. It becomes unreadable.

# Async JavaScript & Callbacks Part 2

In this code, we can see on the left side, there's a triangle like structure to the indentation, this is infamously known as THE CALLBACK HELL. So what callbacks includes is:

*In short, every function gets an argument which is another function that is called with a parameter that is response from the previous one.*

You will definitely get baffled with this sentence, which describes the CALLBACK HELL.

As in our case, you can only imagine, to display several results how many callback functions we will have to make. It's difficult to manage a lot of callback functions.

# Async JavaScript & Callbacks Part 2

Even if you wrote them yourself, you're going to have a hard time understanding them once you come back to the code after some time!

This pattern of coding (i.e. callbacks) at a large scale is not maintainable and is confusing and also violates the DRY principle and hence is a bad practice to follow.

So to resolve this issue, Javascript introduced the concept of promises. In the next section, you're going to fully understand how promises work, and we're going refactor this code to use promises. Stay tuned!

# Promises

In the last section, we've witnessed the callback hell. Now promises come in to the rescue.

What are promises? They are objects that either return the successfully fetched data, or the error.

Let's try to create one:

```
// Creating of the promise
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('Got the user');

    resolve({ user: 'Adrian' });
  }, 2000);
});

// getting the data from the promise
promise.then((user) => {
  console.log(user);
});
```

As you can see, this is much much easier to read.

# Promises

Keep in mind that during the callbacks example, we didn't even test for the errors, when making requests, sometimes, the data may not come back. The reason can be your internet connection, or maybe you're just fetching the data from the wrong database.

In any case, the fact is, that sometimes, you're not going to receive the data you were looking for. And we need to handle those cases. With promises, it's easy.

We'll just replace the resolve with the reject, and pass our error message.

# Promises

```
// Creating of the promise
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('Got the user');

    // resolve({ user: 'Adrian' });
    // reject('Error');
  }, 2000);
});

// getting the data from the promise
// .then gives us the result of the resolve
// and .catch gives us result of the reject

promise
  .then((user) => {
    console.log(user);
  })
  .catch((error) => {
    console.log(error);
});
```

In the last example, with callback functions, we had a lot of functions.

How would we do that with promises?

# Promises

```
console.log(1);

const fetchUser = (username) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the user');

      resolve(username);
    }, 2000);
  });
};

const fetchUserPhotos = (username) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the photos');

      resolve(['photo1', 'photo2']);
    }, 2000);
  });
};

const fetchPhotoDetails = (photo) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the photo details');

      resolve('details...');
    }, 2000);
  });
};
```

# Promises

```
// not this

const user = fetchUser('test', (username) => {
  console.log(username);

  fetchUserPhotos(username, (userPhotos) => {
    console.log(userPhotos);

    fetchPhotoDetails(userPhotos[0], (details) => {
      console.log(details);
    });
  });
});

// and finally, we would call it like this:

fetchUser('Adrian')
  .then((user) => fetchUserPhotos(user))
  .then((photos) => fetchPhotoDetails(photos[0]))
  .then((detail) => console.log(detail));

console.log(2);
```

This is the same thing that we have above. But notice how much easier this is to both read and write. We never again have to use callbacks.

# Promises

Promises are enabling us to write asynchronous code in a much easier way. Recently, there has been an addition to promises. It's called `async` `await`, we're going to check it out next.

# Async/Await

## What is Async-Await?

Async await is simply an addition to promises, it is an easier and cleaner way to work with promises.

The main advantage of asynchronous functions is that they look and behave more like synchronous functions we're all used to. Because of that, it's easier to work with them.

Lets take a look at a simple example

```
const fetchNumber = async () => {
  return 25;
}

asyncOne().then(result => {
  console.log(result) //should log 25
});
```

# Async/Await

Running the above code logs 25 to the console. This means that the promise was fulfilled then returned. If it had not the `.then()` method could not have worked.

Now we come to the special `await` keyword. The `await` waits for the promise to return a result.

The `await` keyword can only be used inside of an `async` function.

Furthermore usage of the `await` expression does not stop the program from running. Rather it only make the functions block with the `async` keyword wait until a promise is fulfilled.

# Async/Await

To go back to our initial example with callbacks and promises, let's do it with `async/await`. `Async await` still uses promises, but with a nicer syntax.

```
const displayData = async () => {
  const user = await fetchUser('Adrian');
  const photos = await fetchUserPhotos(user);
  const detail = await fetchPhotoDetails(photos[0]);

  console.log(detail);
};
```

Look how clean and simple it looks. Amazing!

**CHAPTER 13**

# **Modern JavaScript from ES6 to ES2020**

# ES6+ JavaScript

In this section, we'll explore ES6. So... What exactly is ES6? Is it going to be hard? No, absolutely not! As a matter of fact, you're already writing ES6 JavaScript code.

ECMAScript 6 or simply ES6 is the 6th edition of the set of standards specified by ECMA International to standardize Javascript. In simpler terms, ES6 contains various features. There have been more than 10 editions of ECMAScript with the latest release in 2020 but ES6 remains the most significant and widely implemented edition.

You can think of it in this way: ES6 is to Javascript as your favorite update is to your favorite game. It is simply an "update", to the core functionalities of the language.

# ES6+ JavaScript

But what exactly did it bring, you may ask. So, let's get into it!

## 1. **const and let**

You're already a master at these! In this eBook I am to teach you only the best practices of a language. So from the first chapter, when we started learning variables, you immediately learned about `const` and `let`.

Before ES6 we had only `var`. `var` can still be used, but `let` and `const` make code easier to read and understand as they signal how a variable is used.

Let's see what the differences are.

# ES6+ JavaScript

## Example:

When using var you can reassign your variable. So, you could be creating two variables with the same name twice without knowing it. let doesn't permit that if you are in the same block:

```
var age = 27;
console.log(age); // 27
var age = 28;
console.log(age); // 28

let age = 27;
console.log(age); // 27
let age = 28;
console.log(age); // SyntaxError
```

# ES6+ JavaScript

We can also use const to specify that the variable isn't going to be changed.

```
const password = '123123';
password = '123456'; // TypeError: Assignment to a constant variable

let password = '123123';
password = '123456'; // Allowed
```

## 2. Arrow Functions

The second ES6 addition on the list are our beloved arrow functions. We've used them a lot throughout the course and by now you should be really familiar with them.

# ES6+ JavaScript

Arrow functions allowed us to change our code from:

```
function multiply(x) {  
    return x * x;  
}
```

...to:

```
const multiply = (x) => {  
    return x * x;  
};
```

We can even skip the brackets if we only have one return statement. So, the most compact version of this arrow function would be:

```
const multiply = (x) => x * x;
```

# ES6+ JavaScript

## 3. Default Parameters

The third on our list are default parameters.

With default parameters, you do add default values to your parameters.

For example, if a function has arguments x, y, & z, and when you call this function, you do not mention the value of z.

ES6 assumes its default value which has been given while creating function.

```
const add = (x = 1, y = 2, z = 10) => {
    return x + y + z;
};
```

# ES6+ JavaScript

These are the default argument values which will be used if no value is specified while calling the function.

```
add(10, 3); // 23  
  
// x = 10;  
// y = 3;  
// z = 10; (since it is not defined)
```

## 4. Template Strings

What if I told you that you can add a variable or an object information directly as a string?

Well, you'd say, of course we know that Adrian, we've already learned template strings :)

# ES6+ JavaScript

Template strings are a huge addition to JavaScript, we've worked with them throughout the entire course, you should be familiar with them by now. And you guessed it, they were also introduced in ES6.

For example, let's say that we want to show our customer the order details on our website.

How do we do it?

```
const customer = 'John';

const order = {
  name: 'iPad',
  price: 1400,
};

// the old way ... ugly
const message = 'Hello' + customer.name + ',' + 'do you want to buy ' + order.product + ' for' + order.price + 'bucks?';

// the new way
const message = `Hello ${customer.name}, do you want to buy an ${order.product} for ${order.price} bucks?`;

// That is so simple!
```

# Imports and Exports

This is often used in React.

● ● ● JS dogs.js

```
const dogs = [ 'Bear', 'Fluffy', 'Doggo' ];

const woof = (dogName) => console.log(`${dogName} says Woof!`);

const number = 5;

export dogs, woof, number;
```

● ● ● JS test.js

```
const onlyOneThing = 'test';

export default onlyOneThing;
```

● ● ● JS index.js

```
import { dogs, woof, number } from './dogs.js';

console.log(dogs);
console.log(number);

woof(dogs[0]);

import onlyOneThing from './test.js';

console.log(onlyOneThing);
```

# Spread & Rest

Spread syntax allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

```
function sum(x, y, z) {  
  return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
  
console.log(sum(...numbers));  
// expected output: 6  
  
console.log(sum.apply(null, numbers));  
// expected output: 6
```

# Spread & Rest

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic functions in JavaScript.

```
function sum(...theArgs) {  
  return theArgs.reduce((previous, current) => {  
    return previous + current;  
  });  
  
  console.log(sum(1, 2, 3));  
  // expected output: 6  
  
  console.log(sum(1, 2, 3, 4));  
  // expected output: 10
```

# Array Destructuring

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

If we want to extract data using arrays, it's quite simple using destructuring assignment. Let's refer to our first example for arrays. Instead of going through that repetitive process, we'll do this.

```
const introduction = ['Hello', 'I', 'am', 'John'];
const [greeting, pronoun] = introduction;

console.log(greeting); // "Hello"
console.log(pronoun); // "I"
```

# Object destructuring

The destructuring assignment also works with objects.

We should have an existing object on the right side, that we want to split into variables. The left side contains an object-like “pattern” for corresponding properties. In the simplest case, that's a list of variable names in {...}.

```
const person = { name: 'John', country: 'America', job: 'Developer' };

const { name, country, job } = person;

console.log(name); // "John"
console.log(country); // "America"
console.log(job); // "Developer"
```

**CHAPTER 15**

# **Additional Resources:**

# Resources

It might also be a good idea to do a little practicing before moving on.

If you want something fresh to work on, now would be a fine time to do some coding exercises from across the net.

The following sites are all great places to look:



**LeetCode**

🔗 <https://leetcode.com/>



**Coderbyte**

🔗 <https://coderbyte.com/>

# Resources

## 30 seconds of code

 <https://30secondsofcode.org/>

## Exercism

 <https://exercism.org/>

## Codewars

 <https://codewars.com/>

## Frontend Mentor

 <https://frontendmentor.io/>

# Resources



## Loading.io

Free animated, static spinners, progress bar, backgrounds, and much more.

🔗 <https://loading.io/>



## Lottie Files

High quality, interactive open-source animation files.

🔗 <https://lottiefiles.com/>

# Images

## Unsplash

 <https://unsplash.com/>

## Pexels

 <https://pexels.com/>

## Pixabay

 <https://pixabay.com/>

## StockSnap

 <https://stocksnap.io/>

# Illustrations



🔗 <https://freepik.com/>



🔗 <https://storyset.com/>



🔗 <https://undraw.co/illustrations>



🔗 <https://icons8.com/illustrations>

# Icons



## Font Awesome

🔗 <https://fontawesome.com/>



## Material Icons

🔗 <https://fonts.google.com/icons>



## Flaticon

🔗 <https://flaticon.com/>



## Heroicons

🔗 <https://heroicons.com/>

# Freelancing

 Upwork

 <https://upwork.com/>

 Freelancer

 <https://freelancer.com/>

 Fiverr

 <https://fiverr.com/>

 Toptal

 <https://toptal.com/>

# Hosting

## ◆ Netlify

🔗 <https://netlify.com/>

## ▲ Vercel

🔗 <https://vercel.com/>

## GitHub Pages

🔗 <https://pages.github.com/>

## Firebase

🔗 <https://firebase.google.com/>

# Challenges

The developer's life is a life of challenges (& bugs, haha). To keep up with trends, we have to challenge ourselves constantly. For that, use:

## {XX} CSS Battle

Battleground to improve, tighten CSS skills

 <https://cssbattle.dev/>

## Frontend Mentor

Develop, practice & compete to code a frontend website from its design file.

 <https://frontendmentor.io/>

# Guides

Covers valuable cheat sheets, guides and great tutorials on tech stuff.



## DevDocs

Pragmatic information on almost every language.

<https://devdocs.io/>



## DevHints

Beautiful cheatsheets comprising many languages.

<https://devhints.io/>

# Guides

## GitSheets

A dead simple git cheatsheet.

 <https://gitsheet.wtf/>

## 30seconds of code

Provides short code snippet mainly featuring JavaScript.

 <https://30secondsofcode.org/>

## OverAPI

A site collecting all the cheatsheets, all!

 <https://overapi.com/>

# Tools

Many prominent developers have created excellent, crazy tools to boost development.

These are some of them:



## Remove BG

An AI tool to help remove the background of an image in 5 seconds with just one click.

 <https://remove.bg/>



## Clippy

Creates complex shapes in CSS using clip-path property like (circle, ellipse, polygon, or inset),

 <https://bennettfeely.com/clippy/>

# Tools

## ● Haiei

Generates unique SVG design assets

🔗 <https://haiei.app/>

## 〈/〉 Web Code Tools

One Platform with many solutions. Generates box shadows, gradients, meta tags, etc.

🔗 <https://webcode.tools/>

## ↔ Transform

A polyglot web converter.

🔗 <https://transform.tools/>

# The End.

Congratulations! You've made it to the end.

Thank you so much for reading this book.

Our goal is to update this book with new content as JavaScript evolves.

Now that you're a master of JavaScript, join our journey to master the most in-demand technologies of the modern web development world:



<https://jsmastery.pro>



Don't forget to follow JavaScript Mastery

