NODE.JS ESSENTIALS

A SIMPLE UNDERSTANDING OF JAVASCRIPT AND NODE.JS

ADNAN BABAKAN

Contents

About

This book will help you step by step to start working with Node.js. The main goal in this book is to keep it as concise and straightforward as possible, so it is comprehensible for everyone. As I'm writing this book, I'm quarantined due to the COVID-19 pandemic, and I hope things get better soon. This book is a work of hard dedication, and I hope you enjoy it while reading as much as I enjoyed while writing and always remember being a good programmer is neither memorizing all the codes and syntax nor following every idea without questioning it! It is about being able to solve a problem efficiently, so don't be ashamed of opening this book or other books or even a documentation on the internet thousands of times to check out a description and question it if you think it is wrong. You can use the information below in case you wanted to contact me:

+98 939 313 8160

adnanbabakan.personal@hotmail.com

- *Adnan Babakan*

What is in this book?

This book includes all the basic concepts of JavaScript to get started as well as Node.js code samples. It will walk you through some popular approaches of Node.js. The main goal of this book is to be as concise as possible while explaining everything, so it doesn't make the reader confused or overwhelmed. While reading, this book provides you with some exercises which can help you understand what you've read previously.

Is this book for you?

As mentioned, this book is for whom those want to write Node.js applications or use Node.js as their build tool. Although this book teaches JavaScript, the dialect of JavaScript used in this book mostly will be ES6, and this version might be incompatible with browsers as I'm writing this book, and some codes might not work in browsers. You can use this book to learn JavaScript to design front-end applications as well since JavaScript is mostly the same for both goals, but the main goal here will be back-end.

Chapter 1: Getting Started

What is programming?

This might be the first book you are reading about JavaScript or programming at all. The term **“Programming”** might seem scary for people who are getting started, but programming is just putting instructions in a chain, so a computer knows what to do. I believe this is the best definition of programming. There are lots of programming languages of many types, and JavaScript is one of them. You are not limited to learn one of them, and you can learn almost all of the popular ones if you spend a reasonable amount of time. But keep in mind that knowing a specific programming language deeply and understanding it well is better than knowing ten of them while you only scratched the surface. Programming highly bounds with mathematics. This shouldn’t scare you if you are not good at mathematics because when saying that programming is highly bound with mathematics, it doesn’t mean that you should be a mathematician, what you need is to understand the logic of mathematics and steps you take to solve a problem. Just like in mathematics, when you are solving a problem by programming, you need to divide it into smaller problems and solve them one by one and finally gather them up and get a full result. Programming will help you improve how you see the world!

Everyone should know how to program a computer because it teaches you how to think. – Steve Jobs

What is Node.js?

Node.js is a program that helps you run JavaScript on the server and outside of a browser. So Node.js is not a programming language, and the language that is going to be used in this book is JavaScript (sometimes mentioned/abbreviated as JS). It is assumed that you don’t know JavaScript, so this book will walk you through JavaScript and Node.js simultaneously and step by step.

What is JavaScript?

JavaScript is an object-oriented programming language that is dynamically typed and interpreted. **Dynamically typed** term refers to the unnecessity of defining the data type of variables or arguments. Interpreted means that JavaScript won't be compiled into machine code and will be executed by its interpreter line by line from top to bottom. JavaScript files are suffixed with **.js** and **.jsm** extensions while **.js** is used more often. JavaScript was originally created to be used in the browsers to enhance the web experience, but thanks to Node.js, it can be run on the server-side. JavaScript is the same language with the same syntax in both browsers and Node.js servers, but the tools might vary, and some functions might only be available in the browser instance and some only in Node.js.

JavaScript is a single-threaded programming language meaning that codes are executed one by one, and no two scripts can be run at the same time.

What is EcmaScript?

JavaScript has many versions, and they are referred to as ES5, ES6, etc. ES stands for EcmaScript. This is the definition of EcmaScript, according to Wikipedia:

> *ECMAScript (or ES) is a scripting language specification standardized by Ecma International. It was created to standardize JavaScript to help foster multiple independent implementations.*

Installation

There are several ways of installing Node.js, depending on your operating system. Here we will discuss three of the most used desktop operating systems, which are Microsoft Windows, macOS, and Ubuntu (Debian). Node.js provides two versions at its website to download; one is called LTS which stands for "Long-Term Support" which means it is going to be supported and actively maintained for 12 months as the documentation says, the other version is the latest version of Node.js that ships with the most cutting-edge features that recently got added to Node.js but it is less stable compared to LTS version. So, I highly suggest you download the LTS version since you might get in trouble with the latest version and experience strange and unexpected behavior from your codes.

Windows and macOS

Windows and macOS users can download the setup file from Node.js' official website at https://nodejs.org and install it by running the setup wizard.

Ubuntu (Debian)

To install Node.js on Ubuntu, keep in mind that you'll need sudo privileges and the dollar sign ($) means your command line and isn't a part of the command:

After running the command above you need to install Node.js by running the command below:

```
$ sudo apt install nodejs
```

To make sure that your Node.js instance is up and running, you can check its version by the command below:

```
$ node -v
```

This should show you the current version of Node.js that is installed on your machine. If you receive an error, that means Node.js isn't installed properly.

You'll need an IDE (Integrated Development Environment) to write your Node.js app at its finest. There are several IDEs you can use. The most used ones are VSCode from Microsoft, which is free and WebStorm from JetBrains, which is a paid one. In case you wonder what an IDE is, it is a tool that helps you write, run and debug your code in one place without having to work with a separate text editor, terminal, and some debugging tools. It is not a must to use an IDE but is highly recommended.

If you are not using an IDE, you can run your code from your terminal/cmd by using the node command.

```
$ node app.js
```

Your file name can be anything.

Chapter 2: Starting with JavaScript

Every programming language demands some syntax and logic, and JavaScript is no exception. JavaScript is a relatively easy language to adopt compared to some other languages like C/C++ or Java. JavaScript has some main concepts like variables, constants, control flows (conditions, switch cases, and loops) like almost all other languages and a unique concept of prototypes.

JavaScript is an object-oriented programming language (referred to as OOP as well) and is case-sensitive. Almost everything in JavaScript is an object, keep this in mind even if you don't understand the meaning right now since it is going to help you with further aspects of JavaScript and will make them easier for you to understand.

Statements and keywords

A computer program is a list of "instructions" to be "executed" by a computer.

This is how W3Schools describes a computer program. So we can conclude that a programming language is a tool that helps us write these instructions. Defining a variable, solving a mathematical problem such as 2 + 2, telling JavaScript to print "Hello World" all are examples of these instructions. Instructions in JavaScript are usually categorized as two things: Statements and expressions. Statements are usually the instructions resulting in an action done while expressions usually result in a value.

Statements are often started with keywords to let JavaScript know what the intent of this statement is. Keywords are special words that mean something specific and cannot be redeclared or used in another way other than how JavaScript defined it. Keywords are often referred to as reserved words as well. The table below shows all the keywords:

abstract	arguments	boolean	break	byte	case	catch
char	const	continue	debugger	default	delete	do

double	else	eval	false	final	finally	float
for	function	goto	if	implements	in	instanceof
int	interface	let	long	native	new	null
package	private	protected	public	return	short	static
switch	synchronized	this	throw	throws	transient	true
try	typeof	var	void	volatile	while	with
yield						

Along with these keywords, there are newly added keywords since ES5:

class	enum	export	extends	import	super

As mentioned, these words are meant to do something specific so they cannot be used to name variables, function, and any other thing.

Statements end with a semicolon in JavaScript. Having semicolon is not a must if you have only one statement per line as below:

```
console.log("A")
console.log("B")
console.log("C")
```

But if you want to include all of these in one line, you should use a semicolon to let JavaScript know where each of your statements ends.

```
console.log("A"); console.log("B"); console.log("C")
```

Comments

Comments are parts of code ignored by the interpreter and are not executed. They are only there for humans to read them and know what the code does or write a note to take action later. There are two types of comments in JavaScript, a **single-line comment** and a **multi-line comment**. Single-line comments start with **//,** and all the characters after this sign are commented till the end of the line.

```
console.log('Hello World'); // This is a comment
```

On the other hand, a multi-line comment is different since it can take from one line of comment to how many you needed. Multi-line comments start with **/*** and end with ***/,** and anything between these two signs is considered as comments.

```
/* Hi I'm a multi-line comment
console.log('Hello World');
Although the code above is a valid JavaScript statement,
it won't be executed because it is inside of a comment. */
```

Variables and data types

JavaScript syntax defines two types of values, **fixed** and **variable** values. Fixed values are called **literals,** and variable values are called **variables**. Variables are simply containers to hold our data so we can use or change them later. As in mathematics, each variable can hold a value that is used in other places, and when its value changes, it means all other references of the variable take the new value as well. On the other hand, literals are just what they tend to be, a number is a number, and its identity cannot be altered. Every data in JavaScript belongs to a data type and derives from its prototypes.

The **typeof** keyword can determine data types. In other words, a data type is an answer to the **typeof** keyword.

```
console.log(typeof 'Hello World') // string
console.log(typeof 2) // number
console.log(typeof 5.7) // number
console.log(typeof true) // boolean
```

Primitive data types

Primitive data types are the most basic data types in JavaScript, which are not objects. They consist of **undefined**, **Boolean**, **Number**, **BigInt**, **String**, **Symbol**, and a special type called **null**.

undefined

“undefined” is a type that is assigned to a variable that just got declared with no value automatically.

Boolean

A “Boolean” is a logical data type that can only be one of the **true** or **false** (case-sensitive) values. Booleans are used to determine the truthy of a condition.

Number

A “Number” is a decimal numeric value that can be both an integer or a float number. For example: 2, -4, 0.5, 6.43, -9.8.

Numbers in JavaScript are an implementation of a double-precision 64-bit binary format which they hold numbers from **$-(2^{53} - 1)$** to **$2^{53} - 1$**. The “Numbers” type has some special, symbolic values in addition to floating-point numbers, which are: **+Infinity**, **-Infinity**, and **NaN** (stands for not a number). An instance of **NaN** type is the answer of dividing **0** by **0**.

In this number system, the number 0 (zero) has to representations, +0 and -0**,** which 0 (with no sign) is an alias for +0. Both of these values are equal, which means if we compare these two numbers like +0 === -0, the result is **true**. But in more precise mathematical calculations, you can see the difference. When you divide a number like 10 by +0 the answer is **+Infinity**. On the other hand, when you are dividing it by -0, the answer is **-Infinity**.

BigInt

“BigInt” acts just like Number, but you can work with bigger numbers. “BigInt”s are suffixed with an “n” at the end, like **4n**, **901392471n**, etc.

String

“String” is a data type that holds a sequence of characters that represents a text. For example: **“Hello World”**, **“Adnan”**. Each “String” is a set of 16-bit elements that each element occupies a position, the first element is at index 0, and the second one is at index 1 and so on. Strings are always surrounded by double-quotes (“), single-quotes (‘), or backticks (`).

Symbol

A “Symbol” is a unique atomic value.

null

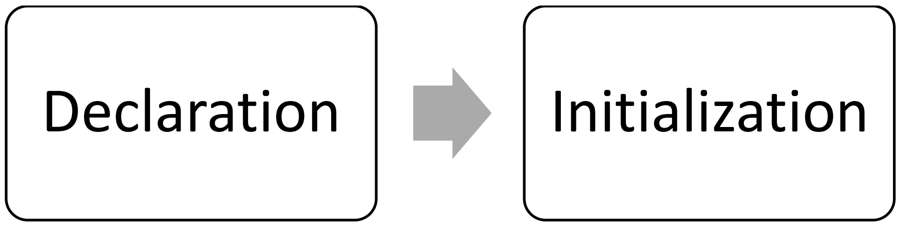
“null” is a special data type that means the term nothing in JavaScript.

Objects

"Object" is another data type in JavaScript, which is widely used but is not primitive. An object is not a data itself but a structure. Almost everything that is made with the **new** keyword is an object, such as: "new Object", "new Array", "new Date".

Defining variables

In JavaScript, a variable is created in two steps, **declaration,** and **initialization**. **Declaration** is the step that you declare your variable's name, and **initialization** is the step that you assign a value to it. So it is always declaration and then initialization.



A variable is declared with the **let** keyword proceeded with your variables name.

When naming your variable, you should consider that:

- Variable names cannot contain spaces
- Variable names must begin with a letter (A-Z a-z), an underscore (_) or a dollar sign ($)
- Variable names can only contain letters, numbers, underscores, or a dollar sign
- Variable names are case-sensitive, means "myName" is different than "MyName"

```
let variableOne;
```

More than one variable can be declared at once like the code below:

```
let variableOne, variableTwo, variableThree;
```

Now all of these variables have the value of **undefined**, so now we want to assign a value to these variables using the assignment operator (=).

```
let variableOne, variableTwo, variableThree;

variableOne = 5;
variableOne = 6;
variableThree = 8;
```

So these three variables are now **initialized** and have values of data types **number**, **boolean,** and **string,** respectively.

Both **declaration** and **initialization** steps can be done at once:

```
let variableOne = 5;
```

You can use **console.log()** to show the value of your variable.

```
let x = 5
console.log(x)
```

You cannot access a variable before initializing it, so the code below will throw an error:

```
console.log(x)
let x = 5
```

Tip: A good practice of naming variables (or functions as well) in JavaScript is following the camel-case convention. Camel-case follows these rules:

- **The first letter of your first meaningful word must be lowercase**
- **Each meaningful word afterward should start with an uppercase letter**

To check the data type of a variable, you can use the **typeof** keyword.

```
let a = 5
let b = "Hello"
let c = true

console.log(typeof a) // number
console.log(typeof b) // string
console.log(typeof c) // boolean
```

Caution: ”typeof” is not a function!

Variables' values can be changed later on:

```
let x = 9
x = 8
console.log(x) // 8
```

Tip: There is another keyword called "var" used for defining variables just like "let". But there are some major differences between these two. Using "var" is now discouraged while programming in Node.js.

Functions

A function is a piece of code that runs as a packaged block of code to make reuse of your codes easier.

Defining constants

Constants are just like variables with one important difference! Their value cannot be changed once it is assigned. A constant is defined using the **const** keyword.

```
const x = 5
console.log(x) // 5
```

You can use a constant anywhere you'd use a variable, but you cannot change its value. Since you cannot change its value, it means both declaration and initialization of a constant should be at once and cannot be separated into two parts, like variables. If you try to change a constant's value, your program will encounter an error:

```
const x = 5
x = 9
```

The code above is not valid and causes an error that stops your program.

Strings

As mentioned before, a string is a sequence of characters chained together. A string is defined using double-quotes, single-quotes, and backticks.

```
let stringOne = "I am a string"
let stringTwo = 'I am a string as well'
let stringThree = `Yet, another string`
```

Strings surrounded by double-quotes and single-quotes are almost the same thing. If you want to use a single-quote inside your string you should surround it by double-quotes, and vice-versa. If you use a single-quote character surrounded by single-quotes, JavaScript won't know where to end your string properly, and that will cause an error.

```
let stringOne = "I'm a string!"
```

As you can see, I used a single-quote as a part of my string. If you wanted to surround it by single-quotes, this is what would happen:

```
let stringOne = 'I'm a string!'
```

In this case, JavaScript won't know where to end your string.

There is also a way of handling such things called escaping. Escaping tells JavaScript to treat this character only as a letter and do not involve it in the syntax. You can escape some characters that they mean something. Here you can escape your single-quote. To escape a character, you need to add a backslash before it.

```
let stringOne = 'I\'m a string!'
```

Although here you see your backslash, if you try to use it somewhere, you will see your string as you expected. If you try to **console.log()** your string, you will see "I'm a string!" printed.

Template literals (template strings)

A string surrounded by backticks is called a template literal or a template string. There is a benefit in using such a thing. Imagine having multiple strings, and you want to concatenate them. What you would do normally is to use the **+** operator:

```
let firstName = 'Adnan'
let lastName = 'Babakan'
let greet = 'Hello ' + firstName + ' ' + lastName
```

So your variable **greet** equals a full sentence like "Hello Adnan Babakan". A template literal can manage this way more easily:

```
let firstName = 'Adnan'
let lastName = 'Babakan'
let greet = `Hello ${firstName} ${lastName}`
```

By using **${VARIABLE_NAME}** syntax inside a template literal, you can embed your variable inside your string.

Code blocks

JavaScript statements can be grouped in code blocks, inside curly braces "{}". The aim of doing such a thing is executing a group of codes together. Code blocks can be found in functions, flow control statements, and some other places. Code blocks will be discussed later on.

Operators

Operators are a crucial part of every programming language. Operators perform an operation on one or two operands. JavaScript has five sets of operators, which are **arithmetic operators**, **comparison operators**, **logical operators**, **assignment operators**, and **conditional operators**.

Arithmetic operators

Arithmetic operators are used for performing mathematical operations.

Operator	**Description**	**Example**	**Result**
+	Sum	9 + 7	16
-	Minus	8 – 3	5
*	Multiply	3 * 4	12
/	Devide	8 / 2	4
**	Power	8 ** 2	64
%	Remaining	11 % 4	3
++	Incrementation by one	let x = 2 x++	3
--	Decrementation by one	let x = 5 x--	4

++ and – operators take action before or after a variable is used according to their position. To understand this aspect, we can simply run a test:

```
let x = 5
console.log(x++)
console.log(x)
```

In this code, on the second line, we first used our variable then incremented it by one, hence the printed result is 5 and then 6.

```
let x = 5
console.log(++x)
console.log(x)
```

But in this one, we first incremented our variable and then used it, so the result includes two 6 numbers printed.

Let’s take a look at a sample of arithmetic operators:

```
let x = 2 + 3 // 5
let y = 5 - 8 // -3
let z = 3 * 5 // 15
let m = 7 / 2 // 3.5
let n = 3 ** 3 // 27
let o = 17 % 5 // 2
```

These operators follow the usual mathematical procedure, which means it is read from left to right, but the multiplier, the divider, and the power are the priority unless they are in parentheses.

Caution: Always remember the "%" operator doesn't mean percentage. If you want to implement percentage in your program, you should use the divider operator "/". For instance "5 / 100" or "90 / 100"

Among all these operators, there is one operator that is commonly used between strings and numbers and that is the **+** operator. When using this operator to concatenate two strings, it is called the **concatenation operator**.

```
let myString = "Hello " + "World" // Hello World
```

As in the statement above, the **myString** variable now equals to "Hello World".

Keep in mind if you use the **+** operator with a number operand and a string operand, the number is considered to be a string, and the result is also a string.

```
let myString = "2" + 4 // 24
```

The variable above equals to 24 but as a string. The simple term is that JavaScript just put the character 2 and 4 together.

```
let myString = "A" + 4 // A4
```

In the example above, we've put the character **A** and 4 together, so the result is **A4**.

Comparison operators

Comparison operators are used for comparing two operands and return a Boolean. The table below describes all the comparison operators available.

Operator	Description	Example	Result
==	Equality without considering the data type involved	2 == "2"	true
		3 == 3	true
		4 == 9	false

		“A” == “A”	true
===	Equality with data type involved	2 === “2”	false
		3 === 3	true
		“A” === “4”	false
		“B” === “B”	true
!=	Inequality without considering the data type involved	4 != 4	false
		4 != “4”	false
		8 != 8	false
		“A” != “C”	true
!==	Inequality with data type involved	4 !== 4	false
		4 !== “4”	true
		“A” !== “A”	false
		“A” !== “B”	true
>	Checks whether the left operand is bigger than the right one	8 > 2	false
		4 > 1	true
<	Checks whether the right operand is bigger than the left one	3 < 4	true
		9 < 2	false
>=	Checks whether the left operand is bigger than or equal to the right one	4 >= 4	true
		2 >= 8	false
<=	Checks whether the right operand is bigger than or equal to the left one	9 <= 0	false
		3 <= 4	true

Caution: New programmers sometimes use the “=” operator instead of “==”. Keep in mind that “=” is an assignment operator and doesn’t compare two operands.

Logical operators

Logical operators are used for combining two or more conditions. The table below describes all the logical operators:

Operator	**Description**	**Example**	**Result**
&&	Known as AND operator as well. Checks whether both operands are true	true && true	true
		true && false	false
		3 > 4 && 4 < 9	false
		9 === 9 && 8 >= 2	true
||	Known as OR operator as well. Check whether	true || true	true
		false || true	true
		3 > 2 || 7 < 2	true

	one of the operands is true at least	5 > 9 || 4 < 1	false
!	Known as NOT operator. It makes the result of the Boolean	!true	false
		!false	true
		!(7 >= 2)	false
		!(true || false)	false

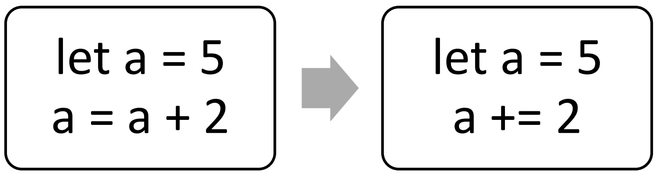
Assignment operators

Assignment operators are used for assigning values to variables. The table below describes all the assignment operators:

Operator	Description	Example	Result
=	Assigns the right side operand to the left operand	let a = 2	--
+=	Sums up the right operand with the current value of the left-hand variable and assigns it to the left-hand variable, can be used for strings as concatenation as well	let a = 2 a += 4	The variable “a” now equals to 6
		let b = “Hello” b += “ World”	The variable “b” now equals “Hello World”
-=	Substracts the right operand from the current value of the left-hand variable and assigns it to the left-hand variable	let a = 10 a -= 7	The variable “a” now equals to 2
*=	Multiplies the right-hand operator’s value by the current value of the left-hand value and assigns it to the left-hand variable	let a = 5 a *= 4	The variable “a” now equals to 20
/=	Divides the current value of the left-hand variable by the operand on the right and assigns it to the left-hand variable	let a = 15 a /= 3	The variable “a” now equals to 5
%=	Get the remaining of the current value of the left-hand variable divided by the right-	let a = 17 a %= 5	The variable “a” now equals to 2

	hand operand and assigns it to the left-hand variable		
**=	Raises the current value of the left-hand variable to the power of the right operand and assigns it to the left-hand variable	let a = 2 a **= 4	The variable “a” now equals to 16

To better understand these operators, lets take a look at how they work to comprehend them with more ease. The assignment operators with an arithmetic operator before the equal sign are just a short way of doing some operations.



```
let a = 5
a = a + 2
```

```
let a = 5
a += 2
```

As you can see in the picture above, the statement **a += 2** is exactly as same as **a = a + 2**, but shorter.

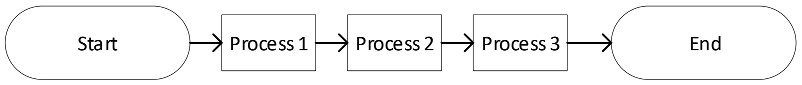
Chapter 3: Structure and Flow Control

Every programming language requires a specific logic to work, and that logic determines the workflow of our program. Workflow means the procedure of our code and how it gets executed.

For instance, as mentioned above, you cannot use a variable before initializing it.

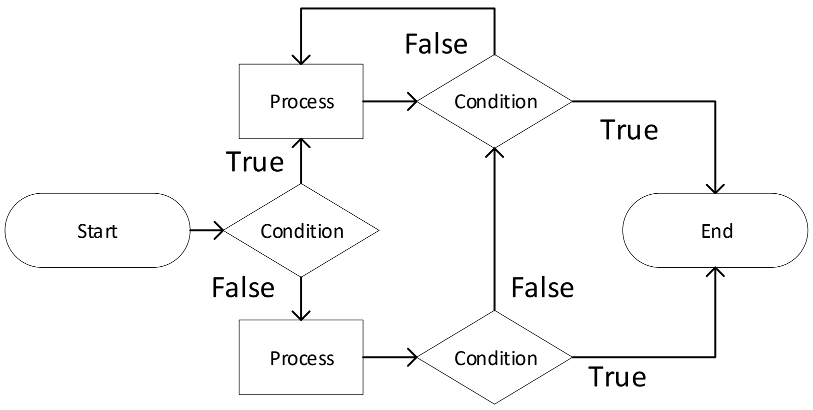
```
console.log(x)
let x = 5
```

So the code above will throw an error since our variable is defined after it is used. You can consider JavaScript's workflow a straight line like this:



Pretty straight forward and simple, JavaScript performs the first process defined and then the second, and so on till it reaches the end (You are going to read about asynchronous programming later in this book, which will defy this rule in some ways).

But that's not how we want our program. A program cannot always be straight forward. It needs some actions taken depending on circumstances. That's where flow control finds a meaning.



As you can see, this flow chart is much more complex, and that is due to some flow-control statements we have. Using flow-control statements helps us build a much more flexible program.

Flow control statements

Flow control statements are kind of statements that change the workflow of our program depending on some conditions.

Conditional statements

A conditional statement executes a block of code if the condition defined evaluates to true. There are three kinds of conditional statements in JavaScript:

- **if** statement
- **if-else** statement
- **if-else if** statement

if statement

An if statement can be defined as below:

```
if (condition) {
    // Code block
}
```

In which your codes inside curly braces will run if the condition is true. Your condition should be a Boolean or something that evaluates to a Boolean.

Let’s have an example:

```
if (3 > 5) {
    console.log('YES')
}
```

If you run the code above, nothing will happen since your condition is wrong. But if you change to the code below, the result will be different:

```
if (8 > 5) {
    console.log('YES')
}
```

Now you will see the word **YES** printed because your condition is true.

You can use logical operators to combine conditions as well:

```
if(3 < 9 || 7 < 1) {
    console.log('YES')
}
```

The code inside the block will run since one of the conditions is true.

```
if(3 < 9 && 7 < 1) {
    console.log('YES')
}
```

But the code inside the block of this condition won’t since both conditions need to be true as we used **&&** operator.

if-else statement

An if-else statement is exactly like an if statement with one more fallback option, so if your condition doesn’t evaluate to a truly value, something else would run:

```
if (condition) {
    // Block 1
} else {
    // Block 2
}
```

Block 1 is executed in case your condition is true, and block 2 if otherwise is the case.

```
if (9 === "9") {
    console.log('YES')
} else {
    console.log('NO')
}
```

In this code, our condition is true because we are checking whether 9 (number) is equal to "9" (string), and the answer is no, they are different. So our second block will run, and the word "NO" will be printed.

if-else if statement

An if-else if statement is a way of having multiple conditions chained together.

```
if (condition1) {
    // Block 1
} else if (condition2) {
    // Block 2
} else if (condition3) {
    // Block 3
} else {
    // Block 4
}
```

The procedure is pretty simple. If condition 1 is true, then block 1 will run; if not, it will proceed to the second condition, and so on, eventually, if none of the conditions are true, the codes inside else block will run.

```
let myFavoriteColor = 'yellow'
if (myFavoriteColor === 'blue') {
    console.log('Your favorite color is blue, like a blueberry!')
} else if (myFavoriteColor === 'red') {
    console.log('Your favorite color is red, like an apple!')
} else if (myFavoriteColor === 'yellow') {
    console.log('Your favorite color is yellow, like a banana!')
} else {
    console.log('I do not know your favorite color')
}
```

In this example, we compared the variable **myFavoriteColor** to some strings to see which one the value of **myFavoriteColor** is equal to. If you run this code, you will see "Your favorite color is yellow, like a banana!" printed.

You can have as many "else if"s as you want.

Tip: You can avoid curly braces if and only if you have just one statement to be executed:

```
if(3 > 5) console.log('YES')
else console.log('NO')
```

Switch statements

A switch statement is used when we are trying to compare a value to many values in our list. And if-else if statement can be useful for this case as well, but if you have hundreds of cases, then it will be unpractical.

The syntax for switch statements is as follow:

```
switch (value) {
    case 'match 1':
        // Code
        break
    case 'match 2':
        // Code
        break
    case 'match n':
        // Code
        break
    default:
        // Code
        break
}
```

The value that we passed to our switch will be matched to every case, and the proper case will be run. If none of the cases matches, then the code defined as default will be run. After each case, a break is necessary but the last one.

```
let myFavoriteColor = 'red'
switch (myFavoriteColor) {
    case 'blue':
        console.log('Blueberry')
        break
    case 'red':
        console.log('Apple')
        break
    case 'yellow':
        console.log('Banana')
    case 'green':
        console.log('Watermelon')
        break
    default:
        console.log('No fruit found!')
}
```

This code will print a sample fruit according to the value of **myFavoriteColor,** and if none is defined, then it will print “No fruit found!”. If you have multiple cases with one common result, you don’t need to repeat yourself and write them again. You can have multiple cases at once:

```
let myFavoriteColor = 'yellow'
switch (myFavoriteColor) {
    case 'blue':
        console.log('Blueberry')
        break
    case 'red':
    case 'yellow':
    case 'green':
        console.log('Apple')
        break
    default:
        console.log('No fruit found!')
}
```

As in this code, you can see, we defined multiple cases with one common code. Literally, before each break, if you define multiple cases, they are combined as one.

Looping statements

A computer program is supposed to do some specific tasks repeatedly. If it is needed only once, then computers become useless. Loops are meant to help you do a task multiple times.

There are three types of loops in JavaScript:

- **for** loops
- **while** loops
- **do-while** loops

for loops

For loops are used for looping for a specific amount of iterations. A for loop syntax is defined as below:

```
for(initializer; condition expression; loop expression) {
    // Code block
}
```

The initializer initializes a counter variable. For most cases and traditionally, this variable is called **i** as in index, but you can name it anything you want as there is no obligation. The conditional expression determines whether our loop has been performed the required number of times; if yes, then the loop stops. And finally, loop expression determines the action to be performed on the counter each time the loop ends.

Let's make a loop to print numbers from 1 to 100:

```
for(let i = 1; i <= 100; i++) {
    console.log(i)
}
```

So basically, we declared a counter variable called **i** then checked if it is still less than or equal to 100 as our condition expression and then incremented its value by one each time.

Now let's define a loop to print odd numbers from 99 to 0:

```
for(let i = 99; i > 0; i -= 2) {
    console.log(i)
}
```

What we've done here is simple, our counter starts from 99, and our loop checks if the counter is still bigger than 0, and each time our counter gets decremented by 2.

Caution: If you define a loop that would never end, for instance, setting the counter 0 and checking if it is still less than 100 and every time while decrementing it instead of incrementing, it will crash your program, and you might even need to restart your computer. These kinds of loops are called infinite loops.

We can relate a loop in programming briefly to Sigma in mathematics. Imagine the formula below:

$$\sum_{n=1}^{20} n = 1 + 2 + 3 + \cdots + 20 = 210$$

This formula can be rewritten as a loop:

```
let sum = 0
```

```
for(let n = 1; n <= 20; n++) {
    sum += n
}
```

The **sum** variable now holds a value of 210.

Or even a more complicated one:

$$\sum_{n=10}^{12} n(n^2+1) = 10(10^2+1) + 11(11^2+1) + 12(12^2+1) = 4092$$

This is the JavaScript code representing the equation above:

```
let sum = 0
for(let n = 10; n <= 12; n++) {
    sum += n * (n ** 2 + 1)
}
```

while loops

While loops are used for the cases when we don’t know when our loop would end. A while loop is defined as below:

```
while (condition) {
    // Code block
}
```

A while loop only accepts a condition. It is up to you to turn the condition to false if needed inside your loop.

Let’s define a while loop to print out numbers from 1 to 100:

```
let i = 1
while (i <= 100) {
    console.log(i)
    i++
}
```

As you can see, we defined our counter outside of our loop and incremented it inside of our loop. If you forget to increment the counter inside the loop, it will make an infinite loop. Using a while loop might seem useless for you now, but there are cases such as reading a stream when you don’t know when the stream ends, and here is where you need a while loop.

do-while loop

A do-while loop is similar to a while loop except for one important fact, in a while loop, the condition is checked even before the first run, so if the condition is not met, the code block won't run at all. On the other hand, if we use a do-while loop, our loop is run at least once since it checks the condition at the end, and if the condition is true, it will move to the next loop. A do-while loop is defined as below:

```
do {
    // Code block
} while(condition)
```

In order to better understand the difference of a while loop and a do-while loop, you can try two loops like this:

```
while(false) {
    console.log('Here')
}
```

If you run this code, you will see nothing printed since the condition is wrong, and our loop won't execute the defined code block.

```
do {
    console.log('Here')
} while(false)
```

But here in this code, although our condition is false, you will see the string "Here" printed once.

Let's have an example of a do-while loop where it prints even numbers from 2 to 100:

```
let i = 2

do {
    console.log(i)
    i += 2
} while(i <= 100)
```

nested loops

A nested loop is not a new type of loop. It means a loop (or more) inside another loop. These kinds of loops are very useful when calculating or iterating through two-dimensional data. For instance, let's write a loop to write all possible points in a 4 by 3 grid:

```
for(let i = 1; i <= 4; i++) {
    for(let j = 1; j <= 3; j++) {
        console.log('x: ' + i + ', y: ' + j)
    }
}
```

The first loop is there to iterate through our **i** counter, and inside it, there is a loop to iterate through our **j** counter. The **i** counter starts from 1 then the **j** goes from 1 to 3, then **i** moves to 2 and **j** goes from 1 to 3 again, so for each **i** there are 3 iterations for **j** which makes a total of 12 prints.

There are two statements used to manipulate the flow of a loop, **continue** and **break**.

continue

A continue statement is used for breaking only one iteration of a loop. For instance, you can print numbers from 1 to 100 using a for loop except for numbers 6 and 53:

```
for (let i = 1; i <= 100; i++) {
    if(i === 6 || i === 53) continue
    console.log(i)
}
```

This code breaks the current iteration of the loop if the counter is equal to 6 or 53 before it reaches the logging statement.

break

A break statement is used for breaking the loop and exiting from it. Unlike the continue statement, this one will make a loop terminate.

```
for(let i = 1; i <= 100; i++) {
    if(i === 10) break
    console.log(i)
}
```

Despite the condition that we defined to loop for a total of 100 times, we defined a condition so if the counter is equal to 10 it would break the loop. So this code will print numbers from 1 to 9.

Labelled statements

A label can be used to name a statement in JavaScript to have more control over them. A label is defined as below:

```
label: statement
```

A label is mostly used for controlling nested loops. Imagine having a nested loop that you want to terminate the outer-loop if a certain action happens inside the inner-loop using the break statement. What you would try to do is as below:

```
for(let i = 1; i <= 10; i++) {
    for(let j = 1; j <= 10; j++) {
        if(j === 3) break
        console.log('x: ' + i + ', y: ' + j)
    }
}
```

But what **break** statement does here is breaking the inner-loop. To solve the problem, you can name your loops and terminate the loop you want by its name:

```
outerloop: for(let i = 1; i <= 10; i++) {
    innerloop: for(let j = 1; j <= 10; j++) {
        if(j === 3) break outerloop
        console.log('x: ' + i + ', y: ' + j)
    }
}
```

As you can see, I named the outer-loop and used its name in front of the **break** statement, so it knows which statement to break. This is also applicable for a **continue** statement.

Chapter 4: Functions and Scopes

Functions are the essence of every programming language and a program. Functions are simply a group of codes that are run together and used for repeating a series of codes without having to rewrite the. You'd come across a term called **DRY** in programming communities, which means **Don't repeat yourself** and functions are one of the few ways of doing so. Every single program consists of thousands of functions chained together to make that program run. A function can both do some stuff and terminate or return a value that can be used in other places. Functions provide almost the same usage as in mathematics. In JavaScript, a function can get as many arguments as you want, but they can only return one value.

Types of functions

JavaScript has few ways of defining functions, which are regular, anonymous, and arrow functions.

Regular functions

A regular or normal function (sometimes referred to as named function) is the most basic way of defining a function in JavaScript. A function is created as below:

```
function myFunction(arg1, arg2, ...) {
    // Code block
}
```

You should start your function with the **function** keyword to let JavaScript know that you want to define a function, then proceed with your functions name, then inside parentheses, you can define your arguments (sometimes referred to as parameters) if there are any and then your code goes inside two curly braces which is called a block.

The part which you define your functions name and arguments is called your function's signature.

Let’s define a function to print a name with some greetings.

```
function greet(name) {
    console.log('Hello ' + name)
}
```

If you write this code and try to run, you will see that nothing happens! This is actually how functions are intended to work. To run a function, you need to invoke (call) it. To invoke a function, you should write its name proceeded with parentheses, and add arguments needed inside those parentheses if there are any.

So here to run our **greet** function, we should call it.

So our final code looks like this:

```
function greet(name) {
    console.log('Hello ' + name)
}

greet('YOUR NAME')
```

Replace **YOUR NAME** with your name and run the code. You should see “Hello YOUR NAME” printed. If you don’t pass a value to an argument, it will be equal to **undefined**.

If you want your function to return a value, you can use the **return** keyword and then the value you acquired.

Let’s create a function that returns the area of a circle:

```
function circleArea(radius) {
    return radius * radius * 3.14
}
```

Now even if you call this function, it won’t show you anything since it is only returning a value. When a function returns a value, that means you can use it as you want, inside other functions, or as a value in other ways. Now to show your circle’s area you can use the **console.log()** function like this:

```
console.log(circleArea(5))
```

As you can see, I used the **circleArea** function as an argument for the **console.log()** function. Now, if you run this code, you can see the number 78.5 printed.

As soon as a function reaches the return statement, the function won't continue anymore, and the codes after it are called unreachable code, meaning that they won't be executed.

```
function circleArea(radius) {
    return radius * radius * 3.14
    console.log('Done')
}

console.log(circleArea(4))
```

The code above will show **50.24** in your command line, and nothing more since the code after the return statement in that function won't be executed.

Given that we have a function in mathematics as below:

$$f(x) = 2x + 1$$

It can be rewritten using a regular function like this:

```
function f(x) {
    return (x * 2) + 1
}
```

If there is a function as below:

$$f(x, y) => R^2 \to R$$

$$f(x, y) = \frac{x + y}{x - y}$$

It can be written in JavaScript as below:

```
function f(x, y) {
    return (x + y) / (x - y)
}
```

Anonymous functions

Anonymous functions are another way of defining functions in JavaScript. As the name describes, they have no name on them and are called by referring to a variable holding them. Anonymous functions are useful when it comes to callbacks (will be discussed later).

This is how to define an anonymous function:

```
let myAnonymousFunction = function(arg1, arg2, ...) {
    // Your codes here
}
```

This is almost as same as a normal function and can be called in the same manner.

Let’s define an anonymous function to calculate the area of a rectangle for us.

```
let rectangleArea = function(width, height) {
    console.log(width * height)
}
```

Now, if you call this function as below, you should see the area of your rectangle.

```
rectangleArea(5, 9)
```

If you want your function only to return a value, you can edit it to the function below:

```
let rectangleArea = function(width, height) {
    return width * height
}
```

Arrow functions

An arrow function is a relatively new way of defining functions in JavaScript, which brings some more advanced topics which will be covered in the next chapters. But as of now, an arrow function is defined in this way:

```
let myArrowFunction = (arg1, arg2, ...) => {
    // Your codes here
}
```

You can omit parentheses if you only have one argument:

```
let myArrowFunction = arg1 => {
    // Your codes here
}
```

But if you are considering defining an arrow function with no arguments, you should have an empty pair of parentheses.

```
let myArrowFunction = () => {
    // Your codes here
}
```

Let’s define an arrow function to print the volume of a sphere:

```
let sphereVolume = radius => {
    let radiusToThePowerOfThree = radius ** 3
    let pi = 3.14
    return 4/3 * pi * radiusToThePowerOfThree
}
```

Now you can use the code below to print the result:

```
console.log(sphereVolume(5))
```

An arrow function can be more concise if you want only to return a quick result, so we can omit our variables and only use a return statement.

```
let sphereVolume = radius => {
    return 4/3 * 3.14 * radius ** 3
}
```

Second, the thing that only an arrow function lets us do is to omit curly braces and the **return** keyword if we only have one statement inside our function and make it as below:

```
let sphereVolume = radius => 4/3 * 3.14 * radius ** 3
```

A very beautiful and concise one-line function to calculate the volume of a sphere!

A mathematical function as below:

$$f(x) = 3x^2.4x + 9$$

Can be written using an arrow function like this:

```
let f = x => (3 * x ** 2) * (4 * x) + 9
```

Arguments object

Each non-arrow function has its own arguments objects. There is a variable of type object available inside every function called **arguments** that hold the argument values.

```
function rectangleArea(width, height) {
    return arguments
}

console.log(rectangleArea(5, 6)) // [Arguments] { '0': 5, '1': 6 }
```

The **arguments** variable is also available as a global variable that returns some information about your current running script, which should not be confused when used inside a function.

Default argument values

Functions can have default argument values meaning that if a value is not passed to the function, it will use the default value defined. A default value can be defined following an equal sign (assignment operator) after the name of the argument:

```
function greet(word = 'Hello', name = 'Adnan') {
    return word + ' ' + name
}

console.log(greet()) // Hello Adnan
```

If you pass a value to the first argument it will use the value you passed to it instead of the default value:

```
function greet(word = 'Hello', name = 'Adnan') {
    return word + ' ' + name
}

console.log(greet('Hi')) // Hello Adnan
```

Anonymous functions and arrow functions can have default argument values as well.

```
let rectangleArea = function(width = 10, height = 10) {
    console.log(width * height)
}

rectangleArea() // 100
```

If you want to define an arrow function featuring default argument values you should wrap your arguments by parenthesis even if there is only one argument.

```
let sphereVolume = (radius = 5) => 4/3 * 3.14 * radius ** 3
```

Recursive functions

Recursive functions are the functions that call themselves over and over until needed. Recursive functions can be defined in all three ways described above, and are useful when it comes to some procedure that needs repeating when loops are not useful.

Let’s define a function to countdown from a number given recursively:

```
function countdown(n) {
    if (n > 0) {
        console.log(n)
        countdown(n - 1)
    } else {
        console.log("BOOM")
    }
}

countdown(10)
```

Our **countdown()** function takes an argument called **n** and checks whether it is more than 0 and if it is so, it shows the number and passes it to itself but subtracted by 1. And this goes on and on till it reaches 0 and then BOOM! Although it seems impossible to call a function inside itself since its definition is not finished yet, it is pretty much possible and very useful. Keep in mind that using a recursive function might take a lot of memory and not be the fastest way of performing an operation repeatedly, so consider using a loop instead of a recursive function if possible.

Scope

Scope is a very important aspect of most programming languages. Scope determines the access level of variables. JavaScript has two types of scopes:

- Local scope
- Global scope

A local scope is where if a variable is defined, it won't be available outside of that scope, but on the other hand, a global scope determines that our variable will be available everywhere.

A block of code defines a scope, and as mentioned above, a function defines a block of code using curly braces, so the variables defined inside a function are local scoped.

For instance:

```
function test() {
    let a = 5
}

test()
```

```
console.log(a)
```

Even though we both defined and called out **test()** function but the variable **a** is not defined outside of our function, so **console.log()** won't be able to access that variable and will throw an error.

```
let a = 8

function test() {
    let a = 5
}

test()

console.log(a)
```

But if you run the code above, you'll see the number 8 printed because the variable **a** is equal to 8 in the scope our **console.log()** exists.

```
let a = 8

function test() {
    let a = 5
    console.log(a)
}

test()

console.log(a)
```

And if you run the code above, you'll see the number 5 printed first and the number 8.

```
let a = 8

function test() {
    a = 5
}

test()

console.log(a)
```

If you run this code, you might think something strange happened, but it is actually correct, and JavaScript hasn't made a mistake! We defined the variable **a** using the **let** keyword outside of our function; then, inside the function, we changed it to 5. Pay attention that we didn't redeclare it since there is no keyword **let** before our variable inside the **test()** function! Scopes work hierarchically, which means it is first local scope if there is any, then global scope, so in this particular case, JavaScript first looked at the inside of our function and didn't find any variable named **a,** so it went one step further and looked at the global scope, and fortunately there was a variable called **a**! So JavaScript took that variable and changed it to 5! That's why we see number 5 if we run the code above. But if we don't call out **test()** function and only define it, our program will print number 8.

Caution: A local scope is prior to the global scope.

If a variable has arguments' names as same as some variables defined globally, the arguments are prior to the globally defined variables.

```
let x = 5
let y = 10

function sum(x , y) {
    return x + y
}

console.log(sum(40, 22))
```

In the code above, although we defined two variables called **x** and **y**, we passed 40 and 22 as our arguments to our function, and they took the place of arguments called **x** and **y,** respectively, so the result of this code is 62 and not 15.

A block of code is not only exclusive for functions, but it also involves loops, classes, and almost everything that defines a code block using curly braces.

For instance, a variable defined inside a loop won't be available outside of it:

```
let i = 0

while (i < 1) {
    let x = 5
    i++
}

console.log(x)
```

Such a code will throw an error since our variable called **x** is not defined outside of our code block and the scope.

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

console.log(i)
```

A code like the one above will throw an error as well. Although our variable called **i** is not defined inside the curly braces, it is still defined as a part of our loop, so the scope is local.

Chapter 5: Data Type Object Wrappers and Prototypes

Almost everything in JavaScript is an object. Every data type in JavaScript is wrapped by an object, providing some functionality when trying to get accessed. All JavaScript objects inherit from a prototype that belongs to its type. A prototype is simply a term to define all properties and methods available for a specific data type. A prototype method can be used by dot notation, as the syntax defined below:

```
yourData.prototypeMethod()
```

And a prototype property as below:

```
yourData.prototypeProperty
```

In this list, you will see the signature of the methods as well. Any argument between brackets (“**[**“ and “**]**”) means they are optional. On the other hand, a method from a data type is called after the data type’s name:

```
dataType.method()
```

String

There are several prototype methods and properties available for strings. Here are the most common ones. A full list of prototypes and methods can be found here:

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

String prototypes

String.prototype.length

This property returns the length of the string in UTF-16 code units. This property is read-only.

```
console.log("Hello World".length) // 11
```

Tip: A prototype method or property can be used through a variable as well.

```
let myString = "Hello World"
console.log(myString.length)
```

String.prototype.charAt(index)

The **charAt()** method returns the character at the specified **index**.

```
console.log("Hello World".charAt(1)) // e
```

Caution: The index of a string starts from 0.

String.prototype.split([separator[, limit]])

This method will split your text by the separator you defined and returns an array.

```
console.log("Hello-beautiful-world".split("-
")) // [ 'Hello', 'beautiful', 'world' ]
```

If you want to split a string by every character, you can define an empty string as the first argument:

```
console.log("Hello".split("")) // [ 'H', 'e', 'l', 'l', 'o' ]
```

The second argument is used for defining a limit, meaning that even if there are matching cases for another split, it will be stopped if the limit is reached.

```
console.log("A-B-C-D-E-F".split("-", 2)) // [ 'A', 'B' ]
```

String.prototype.toLowerCase()

Will return the string converted to lowercase.

```
console.log("HeLLo WoRLd".toLowerCase()) // hello world
```

String.prototype.toUpperCase()

Will return the string converted to uppercase.

```
console.log("heLLo WorlD".toUpperCase()) // HELLO WORLD
```

Number

The number data type doesn't have as many prototypes as the string data type has, but provides many methods to work with numbers. A full list of prototypes and methods can be found here:

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

Number prototypes

Number.isNaN(value)

Determines whether the passed value is **NaN**.

```
console.log(Number.isNaN(2)) // false
console.log(Number.isNaN(NaN)) // true
```

Number.parseInt(string)

Parses an integer from a string. In other words, this method will turn your string to a number.

```
let myString = "57"
console.log(typeof myString) // string
console.log(typeof Number.parseInt(myString)) // number
```

Keep in mind this method will return an integer even if your string represents a float number.

```
let myString = "57.82"
console.log(Number.parseInt(myString)) // 57
```

Number.parseFloat(string)

Parses a float from a string, just like **Number.parseInt()** but also supports float numbers.

```
let myString = "23.64"
let newNumber = Number.parseFloat(myString)
console.log(newNumber) // 23.64
console.log(typeof newNumber) // number
```

Arrays

Arrays are used for storing a sequence of data. Data inside an array can be of any type and can also be different, meaning that you can have an array of multiple elements of different types. An array is defined using two brackets (“**[**“ and “**]**”), and a comma separates each element.

```
let myFavoriteColors = ['blue', 'green', 'red']
```

In order to access each of these specific data inside of your array, you should use the index of that element. An array index starts from 0 and not 1 in JavaScript. This is how you would access your data inside of an array:

```
let myFavoriteColors = ['blue', 'green', 'red']

console.log(myFavoriteColors[0]) // blue
console.log(myFavoriteColors[1]) // green
console.log(myFavoriteColors[2]) // red
```

Your index is surrounded by brackets following your variables name.

An array can hold other data types as well, like numbers, boolean, or any other valid data type.

```
let numberArray = [4, -9, .2, -.16, 6.4]
let booleanArray = [true, false, true, true, false]
```

An array can hold different data types, as well:

```
let mixedArray = ['A string', 4, false]
```

Array data can be assigned later using the index:

```
let petsArray = []
petsArray[0] = 'Dog'
petsArray[0] = 'Cat'
petsArray[0] = 'Iguana'
```

Array prototypes

Array.prototype.length

Returns the number of elements in the array.

```
let coordinates = [90, 50, 12]
console.log(coordinates.length) // 3
```

By setting this prop, you can create an empty array of fixed length:

```
let emptyArray = []
emptyArray.length = 5
console.log(emptyArray) // [ <5 empty items> ]
```

By assigning a length later if you have more elements than the specified length, extra elements will be deleted from the end of the array:

```
let numbersArray = [87, 34, .5, -.7, 99.23]
numbersArray.length = 3
console.log(numbersArray) // [ 87, 34, 0.5 ]
```

Another use case of length can be when you want to access the last item in an array.

```
let numbersArray = [22, 97, 65, 82, 17]
console.log(numbersArray[numbersArray.length - 1]) // 17
```

Using this technique, you are passing the length of the array minus one as the index, which will be the last index of that array given that array indexes start from 0.

Array.ptototype.push()

Adds zero or more items to the array and returns the new array length.

```
let numbersArray = [87, 34, .5, -.7, 99.23]
let count = numbersArray.push(80)

console.log(numbersArray) // [ 87, 34, 0.5, -0.7, 99.23, 80 ]

console.log(count) // 6
```

Array.prototype.pop()

Deletes the last element from an array and return that element.

```
let numbersArray = [87, 34, .5, -.7, 99.23]
let deletedElement = numbersArray.pop()
console.log(deletedElement) // 99.23
console.log(numbersArray) // [ 87, 34, 0.5, -0.7 ]
```

Array.prototype.shift()

Removes the first element from an array and returns that element. Just as opposite of **Array.prototype.pop()** method.

```
let numbersArray = [87, 34, .5, -.7, 99.23]
let deletedElement = numbersArray.shift()
console.log(deletedElement) // 87
console.log(numbersArray) // [ 34, 0.5, -0.7, 99.23 ]
```

Array.prototype.reverse()

Makes an array reverse.

```
let numbersArray = [10, 20, 30, 40, 50]
numbersArray.reverse()
console.log(numbersArray) // [ 50, 40, 30, 20, 10 ]
```

Array.prototype.join()

Attaches the elements in an array to form a string:

```
let numbersArray = [20, 86, 32, 9]
console.log(numbersArray.join()) // 20,86,32,9
```

The delimeter can be specified as well:

```
let numbersArray = [20, 86, 32, 9]
console.log(numbersArray.join('-')) // 20-86-32-9
```

Array.prototype.every()

This method iterates over the items of an array and returns true if all the items meet the condition.

```
let myArray = [90, 28, 67, 334, 12, 45]
let areAllTheItemsGreaterThan5 = myArray.every(function(item) { return item > 5 }
)
```

In the code above, we used a normal function which can be rewritten using an arrow function:

```
let areAllTheItemsGreaterThan5 = myArray.every(item => item > 5)
```

The value returned by this method in this particular code is equal to true since it checks all the values for being greater than 5. But if there is only one item smaller than five, then the value will be equal to false.

```
let myArray = [90, 28, 67, 334, 12, 45]
let areAllTheItemsGreaterThan5 = myArray.every(item => item > 5) // true
let areAllTheItemsLessThan50 = myArray.every(item => item < 50) // false
```

Array.prototype.concat()

This method concats two arrays resulting in one array.

```
let a = [10, 22, 35, 78]
let b = [3, 9, 0]
let c = a.concat(b)
console.log(c) // [10, 22, 35, 78, 3, 9, 0]
```

Array.prototype.some()

This method acts just like the previous method, but it will return true if at least one item meets the condition.

```
let myArray = [90, 28, 67, 334, 12, 45]
let isThereAnyItemLessThan20 = myArray.some(item => item < 20) // true
```

Array.prototype.map()

This method returns a new array with the result of a function applied to each of the items.

```
let myArray = [90, 28, 67, 334, 12, 45]
let multiplyByTwo = myArray.map(item => item * 2)
console.log(multiplyByTwo) // [ 180, 56, 134, 668, 24, 90 ]
```

Array.prototype.filter()

This method returns a new array, including all the items meeting the condition.

```
let myArray = [90, 28, 67, 334, 12, 45]
let moreThan50 = myArray.filter(item => item > 50)
console.log(moreThan50) // [ 90, 67, 334 ]
```

Array.prototype.reduce()

This method is one of the most amazing and useful methods for arrays. As the name describes, this method reduces an array to a single value.

```
let myArray = [90, 28, 67, 334, 12, 45]
let sum = myArray.reduce((acc, value) => acc + value) // 576
```

The first argument of the function this method gets is called accumulation (or total), and the second one holds the current value of the item in the iteration of the array. This method accumulates the value based on what operation is defined. It is possible to subtract as well:

```
let myArray = [90, 28, 67, 334, 12, 45]
let subtract = myArray.reduce((acc, value) => acc - value) // -396
```

If your values are of type string, you can concat them:

```
let myArray = ['A', 'B', 'C']
let concat = myArray.reduce((acc, value) => acc + value) // ABC
```

Concatenating strings of an array in this way is not practical since it is also possible to do so using the **join** method. This example was only mentioned for better comprehension.

Iteration through arrays

Accessing an array's data is simple by using its index, but what if we needed to access all the data within an array? In that case, we should loop through the array to access its elements. There are several ways of doing this, in which the most famous ones are going to be discussed here.

For loop

We can iterate through an array by using its length as the limit of the counter.

```
let myArray = [12, 55, 76, 89, 44]

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

We should use the less-than sign since the length's value is always one unit bigger than the last index.

For/of loop

A for/of loop is almost similar to a for loop but with less syntax.

```
let myArray = [12, 55, 76, 89, 44]

for(item of myArray) {
    console.log(item)
}
```

In this case, you cannot access the index individually, and the only thing you have access to is the element itself.

Array.prototype.forEach()

This method is used to iterate over an array.

```
let myArray = [90, 28, 67, 334, 12, 45]
myArray.forEach((v, i, array) => console.log(i + ' => ' + v))
```

The code above generates the result below:

```
0 => 90
```

1 => 28

2 => 67

3 => 334

4 => 12

5 => 45

The last argument, which in this code is called **array**, is equal to the array itself in case needed.

Spread syntax

The spread syntax is used for expending expressions or elements where there are zero or more arguments (for functions) or elements (for arrays or objects) are expected.

The Spread syntax in functions let you get as many arguments as possible while not defining how many exactly. An argument defined using the spread syntax holds the passed values in an array.

```
function f(...n) {
    console.log(n)
}

f(2, 5, 6, 9) // [ 2, 5, 6, 9 ]
```

Using such a syntax allows you to perform a wide range of actions. Here is a function summing up as many numbers passed to it:

```
function sum(...n) {
    let r = 0
    for(let i = 0; i < n.length; i++) {
        r += n[i]
    }
    return r
}

console.log(sum(6, 8, 10, 19, -2)) // 41
```

And here is one shorter way using arrow functions and the reducer method:

```
let sum = (...n) => n.reduce((acc, v) => acc + v)
console.log(sum(6, 8, 10, 19, -2)) // 41
```

Tip: Although you are defining one argument in this arrow function, you should warp it by parenthesis if you are using the spread syntax.

It is also possible to define arguments before a spread syntax. This makes a function accept a specific number of arguments before treating them using the spread syntax:

```
function calc(operator, ...numbers) {
    switch(operator) {
        case '+':
            return numbers.reduce((acc, v) => acc + v)
        case '-':
            return numbers.reduce((acc, v) => acc - v)
        case '*':
            return numbers.reduce((acc, v) => acc * v)
        case '/':
            return numbers.reduce((acc, v) => acc / v)
    }
}

console.log(calc('+', 3, 4, 5)) // 12
console.log(calc('-', 3, 4, 5)) // -6
console.log(calc('*', 3, 4, 5)) // 60
console.log(calc('/', 3, 4, 5)) // 0.15
```

The spread syntax can be used to expand an array and pass it to a function as different arguments:

```
function f(a, b, c) {
    console.log('a => ' + a)
    console.log('b => ' + b)
    console.log('c => ' + c)
}

let numbersArray = [9, 20, 71]

f(...numbersArray)
```

This function generates the result below:

```
a => 9

b => 20
```

c => 71

Each element inside the array gets in arguments places respectively. But what if we pass more arguments than needed?

```
function f(a, b, c) {
    console.log('a => ' + a)
    console.log('b => ' + b)
    console.log('c => ' + c)
}

let numbersArray = [9, 20, 71, 15]

f(...numbersArray)
```

The rest of them will be ignored in case the function doesn't accept more arguments.

You can merge arrays and object using the spread syntax easily:

```
let a = [1, 2, 3]
let b = [4, 5, 6]
let c = [...a, ...b]
console.log(c) // [ 1, 2, 3, 4, 5, 6 ]
```

Merging objects is the same way, but it got added to JavaScript since ES9, so using it might not be a good idea if not sure which environment is going to be responsible for running JavaScript.

```
let a = { firstName: 'Adnan' }
let b = { lastName: 'Babakan' }
let c = {...a, ...b}
console.log(c) // { firstName: 'Adnan', lastName: 'Babakan' }
```

Objects

In JavaScript, objects are king. If you understand objects, you understand JavaScript.

This is how W3Schools defines objects in JavaScript, and indeed it is not exaggerated, not even a bit. Almost everything in JavaScript is an object. An object can be created using an object literal, using **new** keyword or constructor functions. There is also a new way of defining classes to create objects from which got added to JavaScript in ES6 (2015) but is only a syntactical sugar over JavaScript's already existing prototype-based inheritance.

Object literals

One of the most common ways of defining an object in JavaScript is using object literals. If you've ever worked with JSON (JavaScript Object Notation) data in other programming languages, you should already know how to define object literals in JavaScript. An object contains a pair of keys and values written like **key: value** which they reside inside curly braces. Each key-value pair is separated using a comma from each other.

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}
```

Object properties

The previous code defines an object with three properties. We can access these properties using the dot notation:

```
console.log(myInfo.firstName) // Adnan
```

Or using braces just like in an array:

```
console.log(myInfo["firstName"]) // Adnan
```

They point of using braces over dot notation is that you can use expressions inside braces. For instance, imagine you want to use a value that is stored inside a variable as the key to access a value from your object.

```
let myKey = "firstName"
console.log(myInfo.myKey) // undefined
```

If you try to do such a thing, you'll see **undefined** as your result in the console since there is no property inside your object called **myKey**. If you want to use the value of the variable **myKey** you should use braces instead of dot notation:

```
let myKey = "firstName"
console.log(myInfo[myKey]) // Adnan
```

This time JavaScript will treat **myKey** as an expression and evaluate its value.

Caution: An object's type is an object, but the data inside it can be of any type, even another object. To understand this important fact better, you can use the typeof keyword to test it out.

```
console.log(typeof myInfo) // object
console.log(typeof myInfo.firstName) // string
console.log(typeof myInfo.age) // number
```

Object properties can be changed later using the assignment operator, just like a variable.

```
console.log(myInfo.age) // 20
myInfo.age = 90 // Yeah sure
console.log(myInfo.age) // 90
```

If you try to access a key without declaring it first, you won't get an error, but JavaScript will return it as undefined.

```
console.log(myInfo.id) // undefined
```

You can define a key-value pair inside your object later after declaring your object.

```
myInfo.id = 8
console.log(myInfo.id) // 8
```

You can use the value of a variable as your key when defining a new property, just like accessing:

```
let myKey = 'id'
myInfo[myKey] = 8
console.log(myInfo.id) // 8
```

An object can be of any primitive type, even another object. Objects can get as much complicated as you want them to be.

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20,
    languages: ["Azeri", "Persian", "English", "Turkish", "Spanish"]
}
```

Here I defined a key-value pair containing an array. Its elements can be accessed as below:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20,
    languages: ["Azeri", "Persian", "English", "Turkish", "Spanish"],
    siblings: {
        Tarlan: {
            relationship: 'Sister',
            age: 27
        }
    }
}
```

As another example, I defined an object as a value of one of my properties, which can be used as below:

```
console.log(myInfo.siblings.Tarlan.age)
```

A property can be deleted using the **delete** keyword.

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

delete myInfo.age

console.log(myInfo.age) // undefined
```

In case you need to set the name of an object's property or method from a variable, it should be wrapped in brackets:

```
let propName = 'firstName'
let myInfo = {
    [propName]: 'Adnan'
}

console.log(myInfo.firstName) // Adnan
```

Object methods

So far, we only defined properties that can be of any primitive type, such as strings or numbers, arrays, or even another object. But objects are more than just properties. They can have methods (functions) behaving depending on dynamic data. Imagine having the object below:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName: 'Adnan Babakan'
}
```

And trying to change the **firstName** and **lastName** later, obviously, your **fullName**'s value won't change. You might think you can define it as below, so it changes later:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName: myInfo.firstName + ' ' + myInfo.lastName
}
```

But if you try to access **myInfo.fullName,** you sill get an error since you tried to used **myInfo** before its initialization is completed. To solve this problem you can define a method that returns the data you want:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName: function() {
        return myInfo.firstName + ' ' + myInfo.lastName
    }
}
```

Now by accessing the **fullName** method, you will receive the full name, which consists of the **firstName** and **lastName**.

The reason that this code won't throw an error when you are trying to access the **myInfo** before its initialization is that the function doesn't get invoked yet so the code inside is not evaluated until it is needed to be.

```
console.log(myInfo.fullName()) // Adnan Babakan
myInfo.firstName = 'Tarlan'
console.log(myInfo.fullName()) // Tarlan Babakan
```

By running this code, you will get two different results since the **firstName** got changed, and the **fullName** function works by the current data being in the object, so it acts accordingly.

Instead of using the object's name inside the function, you can use the **this** keyword. The **this** keyword refers to the owner of the function, which in this case is the object holding the function.

Caution: They keyword “this” is not a variable, and you cannot change its value.

Given the explanation above your object can be as below:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName: function() {
        return this.firstName + ' ' + this.lastName
    }
}
```

Using **this** has more advanced benefits, which will be covered later.

A method can be defined in a more abbreviated way by omitting the **function** keyword:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName() {
        return this.firstName + ' ' + this.lastName
    }
}
```

Object methods can have arguments as well:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    fullName() {
        return this.firstName + ' ' + this.lastName
    },
    greet(w) {
        return `${w} ${this.firstName} ${this.lastName}`
    }
}

console.log(myInfo.greet('Hi')) // Hi Adnan Babakan
```

Object getters and setters

Getters and setters are used to get and set data in a way that can be manipulated if required. A getter is defined as below:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20,
    get myAge() {
        return this.age
    }
}
```

Although it looks like a function, you should not include parenthesis when trying to access a getter. The proper way of accessing this getter is as below:

```
console.log(myInfo.myAge) // 20
```

The benefit of defining a getter is that you can manipulate data before letting someone access it.

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20,
    get myAge() {
        return this.age * 2
    }
}

console.log(myInfo.myAge) // 40
```

Defining a setter is also similar to a getter but we use the **set** keyword instead.

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20,
    get myAge() {
        return this.age
    },
    set myAge(n) {
        this.age = n * 3
    }
}
```

Now, if you try to assign a value to **myAge** it will assign to the **age** property but after multiplying it by 3.

```
myInfo.myAge = 30

console.log(myInfo.myAge) // 90
```

A getter and setter can have the same names, but they cannot have the same names as a property's.

Caution: **If a setter or getter uses the same name as a property's even if it is not predefined, it will cause an error of stack size.**

Object constructors

By using object literals, you can only have one instance of that object, which happens to be the variable you assigned your object to. In the previous chapters, we learned the prototyping system that empowers JavaScript. Now here we will define constructors that can be used to create objects using the **new** keyword. An object constructor function is a normal function that uses **this** keyword to assign some properties specifically for an object.

```
function Animal(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
}
```

As a convention, we define constructor functions starting by a capital letter (upper camel case), although there is no difference if you don't, it is a good practice so other programmers would know that it is supposed to be a constructor function.

After defining a constructor function, it is pretty easy to create objects from it. Constructor functions are often called blueprints since they define the structure of the object they are going to create.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)
```

Object properties

Now we can access the properties of this object using the dot notation or using braces just like an object defined using object literal.

```
console.log(animalOne.name) // Gugu
console.log(animalOne['type']) // Iguana
```

Now here we can create another object using our constructor function which will hold the same structure of data but with different data.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)
let animalTwo = new Animal('Jacki', 'Dog', 5)
```

These two objects share the same constructor but different instances of it.

Object methods

It is possible to define a method in a constructor function, so it becomes available for the object instanced from it.

```
function Animal(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
    this.describe = function() {
        return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
    }
}
```

Now, if you create an object using this constructor, you will have access to the **describe** function, which will return a sentence about that animal.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)
console.log(animalOne.describe()) // Gugu is a(n) 2 year-old Iguana
```

These methods can also get some arguments.

Caution: Keep in mind that object constructors don't feature getters and setters like object literals, so the way you can do it is by defining a method.

```
function Animal(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
    this.describe = function() {
        return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
    }
    this.changeAge = function(age) {
        this.age = age
    }
}

let animalOne = new Animal('Gugu', 'Iguana', 2)
animalOne.changeAge(1)
console.log(animalOne.describe()) // Gugu is a(n) 1 year-old Iguana
```

Object prototypes

If you try to add a new property or a new method to an object constructor just like an object literal, you will receive an error. So this makes the code below invalid:

```
function Animal(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
}

Animal.color = 'green'
Animal.describe = function() {
    return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
}
```

Caution: This is different than adding a property or a method to an object. Here we are trying to add it to an object constructor, which is not possible since it is not an object itself but a function. So if you try the code below, it is pretty valid, but the property and the method will only be available for an instance and not to other instances if there are any.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)

animalOne.color = 'green'
animalOne.describe = function() {
    return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
}
```

So if you try to create another instance **colour** property and **describe** method won't be available within it.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)

animalOne.color = 'green'
animalOne.describe = function() {
    return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
}

let animalTwo = new Animal('Jacki', 'Dog', 5)

console.log(animalTwo.describe()) // {{ Throws an error }}
```

Here is the place that the prototyping system helps us add properties and methods to a constructor function.

```
function Animal(name, type, age) {
    this.name = name
    this.type = type
    this.age = age
}

Animal.prototype.color = 'green'
Animal.prototype.describe = function() {
    return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
}

let animalOne = new Animal('Gugu', 'Iguana', 2)

let animalTwo = new Animal('Jacki', 'Dog', 5)

console.log(animalTwo.describe()) // Jacki is a(n) 5 year-old Dog
```

Now both of these objects have a method called **describe** and a property called **color**. The power of the prototype system would be more obvious if you know that defining a prototype can be after creating an object instance.

```
let animalOne = new Animal('Gugu', 'Iguana', 2)

let animalTwo = new Animal('Jacki', 'Dog', 5)

Animal.prototype.describe = function() {
    return this.name + ' is a(n) ' + this.age + ' year-old ' + this.type
}

console.log(animalTwo.describe()) // Jacki is a(n) 5 year-old Dog
```

All JavaScript objects inherit properties and methods form a prototype.

Iteration through objects

Just like arrays, it is possible to iterate through an object as well.

For loop

What you need to iterate through an object is the keys defined in that object, unlike arrays where you only need the index. You can use **Object.keys(obj)** method to receive the keys of an object as an array.

```
let myObj = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}
console.log(Object.keys(myObj)) // [ 'firstName', 'lastName', 'age' ]
```

Now it is possible to iterate through this array and have access to keys and thus to the values respectively.

```
let myObj = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

let keys = Object.keys(myObj)

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

For/in loop

For/in loops are a more convenient way of doing what we did above. Instead of all those steps to access the keys in your loop, you can abbreviate it using a for/in loop.

```
let myObj = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

for(const key in myObj) {
    console.log(key + ': ' + myObj[key])
}
```

So in each time of iteration, there is a constant called **key** you can have access to, which holds the current key.

ES5 object methods

By the release of ES5, some new methods got added to the Object type, which can be used to do so many interesting things. Not all of the methods are mentioned here. You can see all the methods on MDN’s website:

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

Object.defineProperty(object, property, descriptor)

By using this method, you can manipulate the properties of an object with some extra options, like making them immutable.

```
let myObj = {}

Object.defineProperty(myObj, 'name', { value: 'Adnan' })

console.log(myObj.name) // Adnan
```

Defining in this way might seem a bit odd and useless, but if you try to log the whole object, you'll see no properties in it.

```
console.log(myObj) // {}
```

If you already had a property and tried to change its value using this method, it will be visible if you try to log the object:

```
let myObj = {
    name: 'Adnan'
}
Object.defineProperty(myObj, 'name', { value: 'Farideh' })
console.log(myObj) // { name: 'Farideh' }
```

descriptor

Descriptor is an object defining the behavior of the property. There are several options available:

configurable

Defaults to false, if set to true the property defined this way can be mutated or deleted.

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan'})
Object.defineProperty(myObj, 'name', { value: 'Arian'}) // {{ throws error }}
// TypeError: Cannot redefine property: name
```

Or if tried to delete the property it will not affect:

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan' })
delete myObj.name
```

```
console.log(myObj.name) // Adnan
```

But if **configurable** is set as true both mutating and deleting actions of the property is possible:

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan', configurable: true})
Object.defineProperty(myObj, 'name', { value: 'Arian'})
console.log(myObj.name) // Arian
```

And deleting:

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan', configurable: true})
delete myObj.name
console.log(myObj.name) // undefined
```

enumrable

Defaults to false, if set true, the property would be visible during enumeration of the object, for instance, when logging the object or looping through it.

Sample of an object with property defined with enumerable set to false (default):

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan', enumerable: false})
console.log(myObj) // {}
```

And with enumerable set to true:

```
let myObj = {}
Object.defineProperty(myObj, 'name', { value: 'Adnan', enumerable: true})
console.log(myObj) // { name: 'Adnan' }
```

Object.defineProperties(obj, props)

Acts almost like **Object.defineProperty** but can define multiple properties as below:

```
let myObj = {}
Object.defineProperties(myObj, {
    firstName: {
        value: 'Adnan',
        enumerable: true
    },
    lastName: {
        value: 'Babakan'
    }
})
```

```
console.log(myObj) // { firstName: 'Adnan' }
```

Obejct.freeze(obj)

The freeze method prevents the object passed to it from being changed, including its properties and prototypes.

```
let myPet = {
    name: 'Jacky',
    animal: 'Dog',
    breed: 'German Shepherd'
}

Object.freeze(myPet)

myPet.age = 5
myPet.breed = 'Husky'

console.log(myPet) // { name: 'Jacky', animal: 'Dog', breed: 'German Shepherd' }
```

Custom prototypes

JavaScript's prototype-based system empowers the programmer to define a custom method for a specific data type.

```
let names = ['Jake', 'Alex', 'John', 'Gina']
console.log(names.first())
```

This code results in an error since there is no method available for arrays called **first**. To define such a method, we can add it to the prototype of this particular data type:

```
let names = ['Jake', 'Alex', 'John', 'Gina']

Array.prototype.first = function() {
    return this[0]
}

console.log(names.first()) // Jake
```

Or for instance, there is no method available for strings to be reversed. Instead, you should split a string and convert it to an array and then reverse the array and join it again as below:

```
let myString = 'Hello this is my string'
console.log(myString.split('').reverse().join('')) // gnirts ym si siht olleH
```

To avoid doing this task every time you can define a prototype:

```
let myString = 'Hello this is my string'

String.prototype.reverse = function() {
    return this.split('').reverse().join('')
}

console.log(myString.reverse()) // gnirts ym si siht olleH
```

Destructuring assignment (ES6)

The destructuring assignment is a feature that came along with ES6. It is used to unpack values from objects and arrays and assign them to individual variables. It is quite easy to use this feature. Imagine having the array below:

```
let numbersArray = [23, 97, 45, 67]
```

Which you want to have all four of these variables individually stored in a variable. Here is what you might do:

```
let numbersArray = [23, 97, 45, 67]
let first = numbersArray[0]
let second = numbersArray[1]
let third = numbersArray[2]
let fourth = numbersArray[3]
```

Which is perfectly fine! But thanks to the destructuring feature of JavaScript it is possible to do this specific task more conveniently:

```
let numbersArray = [23, 97, 45, 67]
let [first, second, third, fourth] = numbersArray
console.log(first) // 23
console.log(second) // 97
console.log(third) // 45
console.log(fourth) // 67
```

As you can see, defining the variable name just in a pattern that an array is defined assigns the values of the array respectively to the variable names according to their sequence. It is possible to define a default value in case the element doesn’t exist:

```
let numbersArray = [23, 97, 45, 67]
let [first, second, third, fourth, fifth = 100] = numbersArray
```

```
console.log(fifth) // 100
```

It is not mandatory to include all four:

```
let numbersArray = [23, 97, 45, 67]
let [first, second] = numbersArray
```

And it is even possible to skip some of the values and go straight to the one we need by adding empty spaces split by a comma:

```
let numbersArray = [23, 97, 45, 67]
let [,second,,fourth] = numbersArray
console.log(second) // 97
console.log(fourth) // 67
```

Getting the head (the first element of an array) and the tail (rest of the array) is also possible:

```
let numbersArray = [23, 97, 45, 67]
let [head,...tail] = numbersArray
console.log(head) // 23
console.log(tail) // [ 97, 45, 67 ]
```

Destructuring works for objects as well. Given that you have an object as below:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}
```

And you want to do something like this:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

let firstName = myInfo.firstName
let lastName = myInfo.lastName
let age = myInfo.age
```

It is possible to get this task done by using a destructuring assignment:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

let { firstName, lastName, age } = myInfo

console.log(firstName) // Adnan
console.log(lastName) // Babakan
console.log(age) // 20
```

It is also possible to name the variables differently:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
    age: 20
}

let { firstName: fn, lastName: ln, age } = myInfo

console.log(fn) // Adnan
console.log(ln) // Babakan
console.log(age) // 20
```

Just as in arrays it is possible to use a default value when destructuring an object, both when using a custom variable name and using the default one:

```
let myInfo = {
    firstName: 'Adnan',
    lastName: 'Babakan',
}

let { firstName: fn, lastName: ln, age = 20, nation: country = 'Iran' } = myInfo

console.log(fn) // Adnan
console.log(ln) // Babakan
console.log(age) // 20
console.log(country) // Iran
```

JSON

JSON stands for JavaScript Object Notation. It is not exactly a data type, but rather, it is a representation of JavaScript objects in a string format. JSON is widely used in APIs to transfer data between two apps (usually being front-end and back-end). JSON has replaced XML in many APIs due to its better implementations in other programming languages.

Stringify

An action in which you convert your object to JSON is called stringifying. In order to do so, you'll need a function called **JSON.stringify**. This function takes three arguments which its signature looks as below:

JSON.stringify(value[, replacer[, space]])

Value is your object which you want to receive it in string format.

```
let myPet = {
    name: 'Jacky',
    animal: 'Dog',
    breed: 'German Shepherd',
    age: 5
}

let myPetJSON = JSON.stringify(myPet)

console.log(myPetJSON) // {"name":"Jacky","animal":"Dog","breed":"German Shepherd
","age":5}
```

Replacer can be either a function or an array and alters the functionality of stringifying.

If it is a function you can define any condition or any behaviour to filter your data:

```
let myPetJSON = JSON.stringify(myPet, function (key, value) {
    if (typeof value === 'number') {
        return undefined;
    }
    return value;
})

console.log(myPetJSON) // {"name":"Jacky","animal":"Dog","breed":"German Shepherd
"}
```

If you pass an array to the replacer argument your JSON will only include the keys you defined in your array:

```
let myPetJSON = JSON.stringify(myPet, ['name', 'age'])

console.log(myPetJSON) // {"name":"Jacky","age":5}
```

And finally, space argument defines the spacing in your result string. If the spacer is a number it will indent each successive level of the string (max 10) and if it is a string successive levels will be indented by the string.

```
let myPetJSON = JSON.stringify(myPet, null, 5)

console.log(myPetJSON)
/*
{
     "name": "Jacky",
     "animal": "Dog",
     "breed": "German Shepherd",
     "age": 5
}
*/
```

Or:

```
let myPetJSON = JSON.stringify(myPet, null, '\t')

console.log(myPetJSON)
/*
{
        "name": "Jacky",
        "animal": "Dog",
        "breed": "German Shepherd",
        "age": 5
}
*/
```

Tip: "\t" represents the tab character and is a special escaped character, often called a symbol (not to be confused with symbol data type).

Parse

Parsing is the exact opposite action of stringifying, meaning that we can turn a well-formatted JSON string to an object. This functions' signature is as below:

JSON.parse(text[, reviver])

The text argument is the string you want to be parsed into an object:

```
let myPet = JSON.parse('{"name":"Jacky","animal":"Dog","breed":"German Shepherd",
"age":5}')
```

```
console.log(typeof myPet) // object
console.log(myPet.name) // Jacky
```

The reviver argument takes a function which all values will be run through before being parsed. Be sure you return the value unless they won't be present in the result object. For instance, we can log all the values:

```
let myPet = JSON.parse('{"name":"Jacky","animal":"Dog","breed":"German Shepherd",
"age":5}', function(key, value) {
    console.log(value)
    return value
})
```

Symbols

Symbols are quite a new primitive data type. Every symbol is completely unique, meaning that no symbol is equal to any other symbol.

Symbols can be created by the **Symbol** factory function.

```
let x = Symbol()

console.log(x) // Symbol()
console.log(typeof x) // symbol
```

If you compare two symbols or two variables holding Symbol values, it will never evaluate to true.

```
let x = Symbol()

console.log(x === Symbol()) // false
```

When creating a symbol, you can pass a string as the first argument. This value does not affect the uniqueness of the symbol; for better comprehension, take a look at the code below:

```
let x = Symbol('A')
let y = Symbol('A')

console.log(x === y) // false
```

One of the symbols use cases is to use them as object keys. Since they are unique, no other key will be able to retrieve the desired value ever again:

```
let person = {
    firstName: 'Jane',
    lastName: 'Doe'
}

let email = Symbol('email')
let anotherEmailSymbol = Symbol('email')

person[email] = 'janedoes@example.com'

console.log(person[anotherEmailSymbol]) // undefined
console.log(person[email]) // janedoes@example.com
```

Symbols provide a semi-privacy effect as well. For instance, when you are converting your object to a JSON string, you won't be able to see your properties in which their keys are symbols.

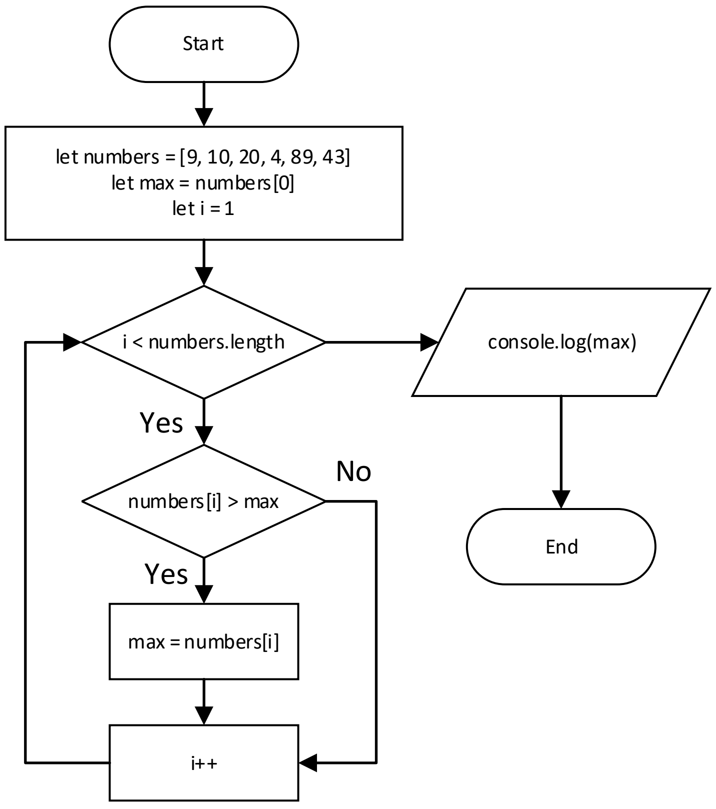
```
let person = {
    firstName: 'Jane',
    lastName: 'Doe'
}

let email = Symbol('email')

person[email] = 'janedoes@example.com'

console.log(JSON.stringify(person)) // {"firstName":"Jane","lastName":"Doe"}
```

In favour of this matter, keys defined with symbols cannot be received using **Object.keys** but instead you should use **Object.getOwnPropertySymbols** method.

```
let person = {
    firstName: 'Jane',
    lastName: 'Doe'
}

let email = Symbol('email')

person[email] = 'janedoes@example.com'

console.log(Object.getOwnPropertySymbols(person)) // [ Symbol(email) ]
```

Exercise: Finding The Largest or Smallest Number

Now that we know all about loops and arrays. Here is a simple problem to solve. How should we find the largest or smallest number within an array? The answer to this problem is pretty simple. Both these numbers can be found in the same way. So here is the flowchart of how to find the largest number in an array:



The algorithm is quite simple. Here is what we do: We consider the first item of the array the maximum value. Then check each other element if it is greater than the current maximum value. If it is so, this value should be the new maximum if not just pass. Finally, what remains is the maximum value in a variable called **max**.

```
let numbers = [9, 10, 20, 4, 89, 43]
let max = numbers[0]

for(let i = 1; i < numbers.length; i++) {
    if(numbers[i] > max) max = numbers[i]
}

console.log(max) // 89
```

Finding the smallest number is quite easy now. We just need to check if the item in the loop is less than the minimum value stored. If it is so, it should be the new minimum value and if not, just pass.

```
let numbers = [9, 10, 20, 4, 89, 43]
let min = numbers[0]

for(let i = 1; i < numbers.length; i++) {
    if(numbers[i] < min) min = numbers[i]
}

console.log(min) // 4
```

Extra: Ternary Operator and Short-circuits

Ternary operator and short-circuits are ways to handle conditions in a more convenient way. Although these features make conditions shorter, they are not usable everywhere and are specific for some situations.

Ternary operator

A ternary operator is used to form a simple one-line if-else condition. A ternary operator's syntax is as below:

```
condition ? true : false
```

If the condition is true, the value after the question mark will be returned. If not, the value after the colon will be returned.

```
let x = 10
let y = x > 5 ? 'YES' : 'NO'
console.log(y) // YES
```

This operator doesn't have any else-if part, but it is super useful for a quick comparison.

Short-circuits

Short-circuits use logical AND (&&) and logical OR (||) operators. By using the logical OR operator, you can set a value or execute a statement if the first value (or following conditions) is not a truly value.

```
let x = false
let y = x || 6
console.log(y) // 6
```

Or with more values:

```
let y = false || 0 || 10
console.log(y) // 10
```

This is especially useful when assigning a default value to a variable if the demanded value is not defined or is false. It is possible to invoke a functio:

```
let x = false
x || console.log('I will run')
!x || console.log('I won\'t run')
```

The logical AND operator acts the opposite of the logical OR operator. In this case, if the value is true, the next value will be placed or invoked if it is a function.

```
let x = true
let y = x && 8
console.log(y) // 8
```

If the first value does not truly value, then it will be placed as the value itself:

```
let x = 0
let y = x && 8
console.log(y) // 0
```

It is possible to have multiple values as well, which all of them have to be true, so the last one gets placed:

```
let x = true
let y = x && !!x && 12
console.log(y) // 12
```

Invoking a function is possible as well:

```
true && console.log('I will run')
false && console.log('I won\'t run')
```

Chapter 6: Arrow Functions In-depth

In chapter 4, we've learned about functions, and a way of defining them called arrow functions. As mentioned, there are some differences between a regular function and an arrow function. Usually, a regular function and an arrow function are interchangeable, but in the case of advanced usage, they provide different approaches.

"This" and arguments binding

An arrow function doesn't have its own **this** and **arguments** bound to it and is derived from the environment it is defined in.

As an example to understand what binding means try the code below:

```
let square = {
    width: 10,
    area: function() {
        return this.width ** 2
    }
}

console.log(square.area()) // 100
```

This code works perfectly since our function is a regular function here. Now, if we convert the **area** function to an arrow function, we will encounter an error.

```
let square = {
    width: 10,
    area: () => {
        return this.width ** 2
    }
}

console.log(square.area()) // NaN
```

As you see the result is **NaN** which means a non-numeric value. For a better understanding let's define two functions to return **this** one being a regular function and the other one an arrow function.

```
let myObj = {
    aString: "Hello World",
    aNumber: 40,
    aBoolean: true,
    regularFunction: function() {
        return this
    },
    arrowFunction: () => {
        return this
    }
}
```

The result of logging the **regularFunction** is as below:

```
console.log(myObj.regularFunction())

// {
//     aString: 'Hello World',
//     aNumber: 40,
//     aBoolean: true,
//     regularFunction: [Function: regularFunction],
//     arrowFunction: [Function: arrowFunction]
// }
```

It returns the object that the function is defined in. But the result of logging the **arrowFunction** will be different:

```
let myObj = {
    aString: "Hello World",
    aNumber: 40,
    aBoolean: true,
    regularFunction: function() {
        return this
    },
    arrowFunction: () => {
        return this
    }
}

console.log(myObj.arrowFunction()) // {}
```

It returns an empty object. The **this** keyword inside an arrow is derived from the environment it is defined in. As in this case, the environment doesn't exist. In other words, **this** keyword is filled with the data out of the scope that the function is located in. It is as if we logged **this** directly:

```
console.log(this) // {}
```

Another example can be a function returning an object including two functions, one being a regular function and the other one an arrow function:

```
function test() {
    this.value = 8

    console.log(this.value) // 8

    return {
        value: 15,
        regularFunction: function() {
            console.log(this.value) // 15
        },
        arrowFunction: () => {
            console.log(this.value) // 8
        }
    }
}
```

There is a value defined inside the **test** function equal to 8. Then it is logged, which obviously prints 8. Later this function returns an object, including three keys. A value, a regular function, and an arrow function. If we log the value inside the **regularFunction** it will print 15 since **this** inside a regular function refers to the object, and inside that object, the value is equal to 15. But if we log the value inside the **arrowFunction** it will print 8 since **this** refers to the environment that the arrow function is defined in, which happens to be the function wrapping this object and returning it.

As such, a regular function has its **arguments** bound to it, but an arrow function refers to the environment in this case as well.

```
function regularFunction(a, b) {
    return arguments
}

console.log(regularFunction(5, 6)) // [Arguments] { '0': 5, '1': 6 }
```

```
let arrowFunction = (a, b) => arguments

console.log(arrowFunction(5, 6)) // [This will print information about the curren
tly running script]
```

Just as the previous example about the **this** keyword, you can try to understand the binding of **arguments** using a function returning an object.

```
function test(a, b) {
    return {
        arrowFunction: (c, d) => arguments
    }
}

console.log(test(6, 7).arrowFunction(9, 10)) // [Arguments] { '0': 6, '1': 7 }
```

As expected, the **arrowFunction** gets the **arguments** from the arguments passed to the parent function, and not the ones passed to itself.

Callable and constructible

In the previous chapter, you've read about object constructors. Object constructors are just regular functions that can be instantiated with the **new** keyword. ES6 distinguishes between functions that are callable and constructible. A regular function is both callable and constructible, but an arrow function is only callable, meaning that it cannot be used with the **new** keyword.

```
let arrowFunction = (a, b) => a * b

let t = new arrowFunction(10, 12)
```

This code will return an error telling that **arrowFunction** is not a constructor.

Chapter 7: Classes (ES6)

Classes are a way to structure objects and are available in most of the object-oriented programming languages like Java, C++, PHP, Ruby, C#, and many more. Unlike Java (or some other programming languages), which in them classes are the base of your program and every file is a class itself, JavaScript doesn't force you to use classes. Classes are not a new thing in JavaScript, and before this chapter, you've already learned classes without knowing. JavaScript classes got introduced in ES6 (ECMAScript 2015) and are just syntactical sugar over JavaScript's prototype-based system. This means JavaScript classes are no different than what you learned in the previous chapter about object constructors and is just a new way of doing the same thing with some benefits.

A class is defined with the **class** keyword followed by the class name; this method of defining a class is called **class declaration**:

```
class Circle {

}
```

There is another way of defining a class, which is called a **class expression**:

```
let Circle = class {

}
```

A class expression can omit the name and can be redefined, unlike a class declaration. As a convention, a class name starts with a capital letter, although there is no obligation.

An instance object of the class can be created by using the **new** keyword just like when using a constructor function:

```
let circle = new Circle
```

Constructor and properties

The main part of a class is its constructor. A constructor function defines the way an object gets created and is called when creating an instance of a class. The usual thing to do in the constructor function is to define the properties of a class (like stats or properties in other programming languages).

Given that our class is representing a circle, its properties will be the coordinates of the centre of the circle and its radius. When declaring or using the properties of a class, you should use the **this** keyword just like in a constructor function.

```
class Circle {
    constructor(x, y, r) {
        this.pointX = x
        this.pointY = y
        this.radius = r
    }
}
```

Now we can create an instance of this class and assign it to a variable:

```
let circle = new Circle(0, 0, 5)
```

Every instance of a class is called an object and holds its own properties. Meaning that each and every object is different than each other even if created from the same class.

```
let circleOne = new Circle(0.25, 0.5, 5)
let circleTwo = new Circle(3, 7, 12)

console.log(circleOne.pointX) // 0.25
console.log(circleOne.pointY) // 0.5
console.log(circleOne.radius) // 5
console.log(circleTwo.pointX) // 3
console.log(circleTwo.pointY) // 7
console.log(circleTwo.radius) // 12
```

Class properties are public and can be accessed or modified outside of the class.

```
console.log(circle.pointX) // 0

circle.pointX = 10

console.log(circle.pointX) // 10
```

Methods

Methods are functions that are attached to the object created using a class. In JavaScript, class methods are prototype methods, just like when creating methods for object constructors mentioned in chapter 6.

So far, our **Circle** class doesn't have any methods, we have to define methods, so it becomes useful. A method can be defined just like a function inside the class, except that it doesn't have the **function** keyword.

```
class Circle {
    constructor(x, y, r) {
        this.pointX = x
        this.pointY = y
        this.radius = r
    }

    area() {
        return 3.14 * (this.radius ** 2)
    }
}
```

Now by using our objects, we can get the area of the corresponding circle.

```
let circleOne = new Circle(0.25, 0.5, 5)
let circleTwo = new Circle(0.75, 5, 12)

console.log(circleOne.area()) // 78.5
console.log(circleTwo.area()) // 452.16
```

It is possible for a method to have arguments just like a normal function:

```
class Circle {
    constructor(x, y, r) {
        this.pointX = x
        this.pointY = y
        this.radius = r
    }

    area(pi = 3.14) {
        return pi * (this.radius ** 2)
    }
}
```

```
let circleOne = new Circle(0.25, 0.5, 5)
let circleTwo = new Circle(0.75, 5, 12)

console.log(circleOne.area()) // 78.5
console.log(circleOne.area(3)) // 75
console.log(circleOne.area(3.14159265359)) // 78.53981633975
```

Here we have defined an argument with a default value representing the pi number.

It is also possible to call a method from another method in case you want to use it:

```
class Circle {
    constructor(x, y, r) {
        this.pointX = x
        this.pointY = y
        this.radius = r
    }

    area(pi = 3.14) {
        return pi * (this.radius ** 2)
    }

    describe() {
        return 'This cirlce is placed at x: ' + this.pointX
            + ', y: ' + this.pointY
            + ' and its area is equal to ' + this.area()
    }
}

let circleOne = new Circle(0.25, 0.5, 5)

console.log(circleOne.describe()) // This cirlce is placed at x: 0.25, y: 0.5 and
 its area is equal to 78.5
```

Getter and setter

Just like objects, a class can have a getter and setter for its properties since classes are just a new way of defining objects after all. A getter or a setter is defined with the **get** and **set** keywords, respectively.

```
const assert = require('assert')

class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName
        this.lastName = lastName
    }

    get fullName() {
        return this.firstName + ' ' + this.lastName
    }

    get myAge() {
        return this.age
    }

    set myAge(n) {
        this.age = n
    }
}

let me = new Person('Adnan', 'Babakan')

console.log(me.fullName) // Adnan Babakan

me.age = 19

console.log(me.age) // 19
```

The rule is also the same, a getter or a setter cannot use the same name as a property's.

Static methods

Static methods special kind of methods which should be called with the class name itself, meaning that it cannot be called through an object. In other words, static methods cannot be called by instantiating the class. These types of methods are usually defined to provide a utility function in a more meaningful way. Imagine you are trying to have methods which calculate the area of different kind of shapes, in this case, you can define your functions inside a class and have them all together. In order to define a static method, you should use the **static** keyword.

```
class Area {
    static square(width) {
        return width ** 2
    }
}
```

The **square** method can now be accessed like this:

```
console.log(Area.square(5)) // 25
```

It is not possible to access this method if an object is created:

```
let shapes = new Area
console.log(shapes.square(5))
```

The code above will log an error.

Now let's complete this class, so it features some function to calculate the area of some shapes.

```
class Area {
    static square(width) {
        return width ** 2
    }

    static rectangle(width, height) {
        return width * height
    }

    static cirlce(radius) {
        return (radius ** 2) * 3.14
    }

    static triangle(base, height) {
        return base * height / 2
    }

    static trapezoid(baseOne, baseTwo, height) {
        return (baseOne + baseTwo) * height / 2
    }

    static parallelogram(base, height) {
        return this.rectangle(base, height)
    }
}
```

As you can see, there are few static methods defined in this class, which they return the area of different shapes. The **parallelogram** method returns the same result from the **rectangle** method as they both use the same formula.

Class fields

Declaration of fields in JavaScript classes adds more readability to your class. Fields are just like properties, but they can be both public and private. A field can be declared without any value assigned to it.

Public fields

A public field is defined inside the class but outside of the constructor function:

```
class Square {
    width = 20
    height
}
```

These fields can be accessed or modified inside or outside of the class:

```
class Square {
    width
    height

    constructor(width, height) {
        this.width = width
        this.height = height
    }

    area() {
        return this.width * this.height
    }
}


let myShape = new Square(22, 20)

console.log(myShape.width) // 22

console.log(myShape.area()) // 440

myShape.width = 78

console.log(myShape.area()) // 1560
```

Private fields

Private fields are the main reason you'll probably use fields. By defining a private field, it can only be accessed inside the body of the class, meaning that it cannot be accessed or altered unless a proper method is defined to do so. A private field is defined with a hashtag prefix, and it can only be declared up-front, outside of the constructor, and cannot be declared later like properties.

```
class Square {
    #width = 20
    #height = 30

    area() {
        return this.#width * this.#height
    }
}
```

So we try to access a private field, it will throw an error:

```
let myShape = new Square

console.log(myShape.#width)
```

But since the **area** function is defined inside the body of the class, it can access the private fields, so calculating the area won't throw an error:

```
let myShape = new Square

console.log(myShape.area()) // 600
```

Private fields' values can be assigned inside a function, like a constructor:

```
class Square {
    #width
    #height

    constructor(width, height) {
        this.#width = width
        this.#height = height
    }

    area() {
        return this.#width * this.#height
    }
}

let myShape = new Square(12, 18)

console.log(myShape.area()) // 216
```

Inheritance

Inheritance is a way to inherit the methods and properties of another class. Inheritance can be useful when you are defining multiple classes with some common behavior. A class that inherits another class is defined using the **extends** keyword.

```
class SubClass extends ParentClass {

}
```

In this example, the **SubClass** class extends the **ParentClass** class. All methods, properties, and fields are inherited except the private fields.

In the example below, we are defining a class called **Animal,** and then we are extending it using the **Dog** class.

```
class Animal {
    age = 1

    walk() {
        console.log('I\'m walking')
    }
}

class Dog extends Animal {

}

let myDog = new Dog

myDog.walk() // I'm walking
```

As you can see, although we are creating an object from the Dog class, which is empty, we have access to a method called **walk,** which is defined in the Animal class. Now we can add other methods and properties to the Dog class, which won't be available if we instantiate an object using the Animal class.

```
class Animal {
    age = 1

    walk() {
        console.log('I\'m walking')
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof Woof!')
    }
}

let myDog = new Dog

myDog.bark() // Woof Woof!

let myAnimal = new Animal

myAnimal.bark() // *This function is not defined*
```

As you can see, the **bark** function is not defined and will throw an error.

Super

Super is a special keyword that is used for calling functions and accessing properties on an object's parent.

When super appears in a constructor function, it calls the constructor of the parent class:

```
class Animal {
    constructor(name, age) {
        this.name = name
        this.age = age
        console.log('I\'m called from the Animal class')
    }
}

class Dog extends Animal {
    constructor(name, age, breed) {
        super(name, age)
        this.breed = breed
        console.log('I\'m called from the Dog class')
    }
}

let dog = new Dog('Kaan', 2, 'German Shepherd')

// I'm called from the Animal class
// I'm called from the Dog class
```

The super method is called in the constructor of the Dog class provided with all the arguments needed for the constructor of the Animal class. As a result, when creating a new object using the Dog class, both sentences are logged, and all the props are set.

The super keyword is not only limited to constructors and can be called within any method inside a class, but the method name should be mentioned following dot notation (it is not needed to mention constructor’s name since a constructor is a special method):

```
class Animal {
    walk() {
        console.log('I\'m waling')
    }
}

class Cheetah extends Animal {
    walk() {
        super.walk()
        console.log('And I am fast!')
    }
}

class Turtle extends Animal {
    walk() {
        super.walk()
        console.log('But I am slow :)')
    }
}

let cheetah = new Cheetah
let turtle = new Turtle

cheetah.walk()
// I'm waling
// And I am fast!
turtle.walk()
// I'm waling
// But I am slow :)
```

By using super you can call static methods as well:

```
class Oval {
    static whatIsAnOval() {
        console.log('An oval is a shape!')
    }
}

class Circle extends Oval {
    static whatIsACirlce() {
        super.whatIsAnOval()
        console.log('A circle is an oval but different!')
    }
}

Circle.whatIsACirlce()
// An oval is a shape!
// A circle is an oval but different!
```

Using “insatcneof”

By using the **instanceof** keyword, we can determine if an object is created from a specific class. This keyword evaluates to a Boolean.

```
class Animal {

}

class Shape {

}

let dog = new Animal

console.log(dog instanceof Animal) // true
console.log(dog instanceof Shape) // false
```

Exercise: Queue and Stack

Queue and stack both are data structures used to handle a collection of data in programming languages. In this section, we will implement the queue and stack using classes in JavaScript as a way to practice using classes and getting to know them. What we are implementing here is a matter of personal design as queue and stack can be designed more or less differently but they have some common approaches that should be considered.

Queue

Queue follows the first-in-first-out principle, often referred to as FIFO. A queue enqueues an item and dequeues it first, meaning that the first item that has been added to the queue will be accessed first.

First, we need to define the **Queue** class in a file called **Queue.js** with a constructor to define the items array.

```
class Queue {
    constructor() {
        this.items = []
    }
}
```

Now we should implement a method to enqueuer the items:

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }
}
```

Now we need a method to dequeue, which means returning the first added item and then removing it from the queue.

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        return this.items.shift()
    }
}
```

But there is a problem here which we have to solve. If there is no item available in the items array, shifting will return undefined. We can handle this by adding a method to determine if the items array is empty and if so is our queue.

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }
}
```

Now that we have this method, we can use it in the **dequeuer** method and return a string telling us what is happening.

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }
}
```

We would need to access the first item in the queue without removing it. We can create a method called **front** to do so. If there is no item in the queue, this method should return a string:

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.items[0]
    }
}
```

We can have a getter method to return the length of the queue:

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.items[0]
    }

    get length() {
        return this.items.length
    }
}
```

Finally, our queue class needs a method to return the items:

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.items[0]
    }

    get length() {
        return this.items.length
    }

    getItems() {
        return this.items
    }
}
```

And a method to prettify the queue items:

```
class Queue {
    constructor() {
        this.items = []
    }

    enqueue(item) {
        this.items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.items.shift()
    }

    isEmpty() {
        return this.items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.items[0]
    }

    get length() {
        return this.items.length
    }

    getItems() {
        return this.items
    }

    prettify() {
        let resultString = ''
        for(let i = 0; i < this.items.length; i++)
            resultString += this.items[i] + ' '
        return resultString
    }
}
```

For the queue to be more secure, you can alter the way the items array is defined. In this code, the items array can be manipulated outside of the class but you can use a private field to prevent this action.

```
class Queue {
    #items = []

    enqueue(item) {
        this.#items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.#items.shift()
    }

    isEmpty() {
        return this.#items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.#items[0]
    }

    get length() {
        return this.#items.length
    }

    getItems() {
        return this.#items
    }

    prettify() {
        let resultString = ''
        for(let i = 0; i < this.#items.length; i++)
            resultString += this.#items[i] + ' '
        return resultString
    }
}
```

Now that we’ve defined the class, we need to export it:

```
class Queue {
    #items = []

    enqueue(item) {
        this.#items.push(item)
    }

    dequeue() {
        if(this.isEmpty())
            return 'Underflow'
        return this.#items.shift()
    }

    isEmpty() {
        return this.#items.length === 0
    }

    front() {
        if(this.isEmpty())
            return 'No item in queue'
        return this.#items[0]
    }

    get length() {
        return this.#items.length
    }

    getItems() {
        return this.#items
    }

    prettify() {
        let resultString = ''
        for(let i = 0; i < this.#items.length; i++)
            resultString += this.#items[i] + ' '
        return resultString
    }
}

module.exports = Queue
```

Here is an example of using the queue to list foods:

```
let Queue = require('./queue')

let foods = new Queue

console.log(foods.length) // 0

foods.enqueue('pizza')
foods.enqueue('hamburger')
foods.enqueue('kebab')
foods.enqueue('kufte')

console.log(foods.length) // 4

console.log(foods.getItems()) // [ 'pizza', 'hamburger', 'kebab', 'kufte' ]

console.log(foods.prettify()) // pizza hamburger kebab kufte

console.log(foods.dequeue()) // pizza
console.log(foods.dequeue()) // hamburger
console.log(foods.dequeue()) // kebab
console.log(foods.dequeue()) // kufte

console.log(foods.length) // 0
```

Stack

Stack follows the last-in-first-out concept, often referred to as LIFO. Just as the opposite of queue, stack pushes items and pops (retrieves) them in reverse order meaning that each item that has been added recently gets out first.

First, it is needed to define a **Stack** class in a file called **Stack.js** with a constructor to define the items stack.

```
class Stack {
    #items = []
}
```

Then there should be a method to push an item to the items:

```
class Stack {
    #items = []

    push(item) {
        this.#items.push(item)
    }
}
```

While popping we should check if the stack is not empty and for doing so we'd need a method to check this matter:

```
class Stack {
    #items = []

    push(item) {
        this.#items.push(item)
    }

    pop() {
        if(this.isEmpty()) return 'Underflow'
        return this.#items.pop()
    }

    isEmpty() {
        return this.#items.length === 0
    }
}
```

Next, we'll add two other methods called **getItems**, **prettify** along with a getter method called **length** just like the **Queue** class. And finally, the **Stack** class should be exported:

```
class Stack {
    #items = []

    push(item) {
        this.#items.push(item)
    }

    pop() {
        if(this.isEmpty()) return 'Underflow'
        return this.#items.pop()
    }

    isEmpty() {
        return this.#items.length === 0
    }

    get length() {
        return this.#items.length
    }

    getItems() {
        return this.#items
    }

    prettify() {
        let resultString = ''
        for(let i = 0; i < this.#items.length; i++)
            resultString += this.#items[i] + ' '
        return resultString
    }
}

module.exports = Stack
```

Here is a sample of using the stack class we created:

```
const Stack = require('./Stack')

let foods = new Stack

foods.push('pizza')
foods.push('hamburger')
foods.push('kebab')
foods.push('kufte')

console.log(foods.length) // 4

console.log(foods.getItems()) // [ 'pizza', 'hamburger', 'kebab', 'kufte' ]

console.log(foods.prettify()) // pizza hamburger kebab kufte

console.log(foods.pop()) // kufte
console.log(foods.pop()) // kebab
console.log(foods.pop()) // hamburger
console.log(foods.pop()) // pizza

console.log(foods.length) // 0
```

Caution: It might be confusing since the results of both queue and stack samples are similar, but you should pay attention to the fact that in the queue items were accessed by the order they were added to the queue. In the other hand, items were accessed from the last one to the first one in reverse order when usuing the stack implementation.

Chapter 8: Code Splitting and Modules

Imagine when you are assigned to write a project that might take more than 10000 lines! Writing all the code in one file is practically not a good idea for a few reasons. First, you have to search for a particular code you wrote inside your file, which can be done much faster if you split your code into files with related names, second debugging your code will be painful since you have to search for all the code again. Creating a program can be a real headache if we try to reinvent the wheel every time. Even the greatest programs consist of codes the programmer didn’t write them himself/herself. Imagine when you want to use a date in your program, you can rewrite a whole calendar system, but this is going to take you much more time compared to if you decided to use someone else’s implementation. Or imagine writing a function over and over again while you can save it in one place and call it from everywhere.

Every programming language has a feature to import other codes. The codes that are getting imported are usually called modules, libraries, or packages depending on the concept and the programming language we are using. Node.js is no exception and allows you to import your own code or a module.

Export/Import your own code

Here we’ll write a function to greet us in the language of our country and export it to be usable in other files.

First, create a file named **greet.js** in the same directory as your **app.js** or whichever file you are running to test your codes. Then create a function called **greet** (doesn’t have to be equal to the name of the file) that gets two arguments, one being the name to be greeted and the second the nationality of that person, which has a default value:

```
function greet(fullName, nationality = 'Iranian') {

}
```

Then by using switch-case, we can decide what to return according to the nationality and choose the English language as the default case if no case matches the **nationality** argument:

```
function greet(fullName, nationality = 'Iranian') {
    switch(nationality) {
        case 'Iranian':
            return 'سلام ' + fullName
        case 'English':
        case 'American':
        default:
            return 'Hello ' + fullName
        case 'Spanish':
            return 'Hola ' + fullName
        case 'German':
            return 'Hallo ' + fullName
    }
}
```

Tip: The reason that we didn't use break in this switch is that we return a value, and as you've read before, the code after a return statement is unreachable and won't be executed.

Now to export this function so it can be imported elsewhere, we need to do as below:

```
function greet(fullName, nationality = 'Iranian') {
    switch(nationality) {
        case 'Iranian':
            return 'سلام ' + fullName
        case 'English':
        case 'American':
        default:
            return 'Hello ' + fullName
        case 'Spanish':
            return 'Hola ' + fullName
        case 'German':
            return 'Hallo ' + fullName
    }
}

module.exports = greet
```

The last line will export our function. Almost anything can be exported regardless of its data types, such as object, numbers, strings, and functions.

Now in your **app.js,** create a variable or constant (it is a good practice to use a constant when importing a file or module) and require your file using an absolute path:

```
const greet = require('./greet')
```

Although our file is called **greet.js,** I didn't include the file extension when requiring it because Node.js is wise enough to know that.

Our variable name doesn't have to match the function name we defined in our **greet.js,** but it's better to be so to prevent confusion.

Then we can use our function just as below:

```
const greet = require('./greet')

console.log(greet('Adnan Babakan', 'Iranian'))
console.log(greet('Sergio Sanchez', 'Spanish'))
```

Exporting is not only limited to function, and you can export every data type, but it is not usually useful to export only a string or an integer. What you can do is export an object, including the data needed. This is usually the case when you want to export a config object to use it wherever needed.

```
let config = {
    applicationName: 'My awesome app',
    author: 'me',
    email: 'adnanbabakan.personal@hotmail.com'
}

module.exports = config
```

Exporting can be shortened by exporting what we need directly and not by assigning it to a variable first and then exporting it.

```
module.exports = {
    applicationName: 'My awesome app',
    author: 'me',
    email: 'adnanbabakan.personal@hotmail.com'
}
```

But it is not a good practice to do so since it is good to separate two logic parts of your file.

Importing a module

There are several modules that are built-in or can be installed manually. There are several ways to install a module, but the best practice is to use a package manager like npm. A package manager is a software that helps you organize your dependencies, and npm is the default package manager for Node.js.

To install a module from the npm repository, you can use the command below in your project directory:

```
$ npm i library
```

Which you have to replace the library name by the library you want. This command will install your library locally, meaning that it will only be available for your current project. If you want to install it globally on your computer so it becomes available in every project, you should add a **–g** flag:

```
$ npm i –g library
```

In contrast to importing your own file, when you are importing a module that is built-in, or you have installed it using npm, you shouldn't include it using an absolute path. Instead, you should only require it by its name, the rest of it is done by Node.js.

For instance, here we'll import a built-in module called **os,** which provides us with a bunch of information about the operating system that our program is running on:

```
const os = require('os')

console.log(os.arch()) // x64
```

There are lots of built-in modules that require you to know callbacks; hence you will read more about these modules in the next chapters.

Export/Import (ES6)

ES6 provides us with a new way of exporting and importing our codes and sharing them between files. While we are using Node.js version 12.x, we need to tell Node.js to treat our files as ES6 modules so we can use this particular feature. To do so, create a file called **package.json** (you’ll read about this file in the next chapters) in the root of your project and put the code below in it:

```
{
    "type": "module"
}
```

This configuration tells Node.js that our files should be treated as ES6 modules. There is an alternative way, which is naming files with **.mjs** extension instead of **.js,** but it is not practical. After doing so, we can start exporting and importing using a new keyword, the **export** keyword. It is possible to export variables, constants, functions, and classes by prepending the **export** keyword before defining/declaring them:

```
export let people = ['Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alireza Por
kar']

export const winter = ['January, February', 'March']

export function greet() {
    console.log('Hi!')
}

export class Person {
    constructor(firstName, lastName, age) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    introduce() {
        return `Hi! My name is ${this.firstName} ${this.lastName} and I'm ${this.
age} years old.`
    }
}
```

It is also possible to export them later:

```
let people = ['Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alireza Porkar']

const winter = ['January, February', 'March']

function greet() {
    console.log('Hi!')
}
```

```
class Person {
    constructor(firstName, lastName, age) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    introduce() {
        return `Hi! My name is ${this.firstName} ${this.lastName} and I'm ${this.
age} years old.`
    }
}

export {people, winter, greet, Person}
```

As for importing, you need to import them by the **import** keyword while declaring what you need:

```
import { winter, people } from './testLib.js'

console.log(winter) // [ 'January, February', 'March' ]
console.log(people) // [ 'Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alireza
 Porkar' ]
```

You can define an alias for each of your imports, so they become available by that name later. This is useful when you have multiple libraries with the same name for their variables, constants, functions, or classes:

```
import { people as friends, Person as P } from './testLib.js'

console.log(friends) // [ 'Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alirez
a Porkar' ]

let me = new P('Jane', 'Doe', 30)
console.log(me.introduce()) // Hi! My name is Jane Doe and I'm 30 years old.
```

The same syntax is available for export, where you can define alias while exporting:

```
let people = ['Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alireza Porkar']

const winter = ['January, February', 'March']

function greet() {
    console.log('Hi!')
}
```

```
class Person {
    constructor(firstName, lastName, age) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    introduce() {
        return `Hi! My name is ${this.firstName} ${this.lastName} and I'm ${this.
age} years old.`
    }
}

export {people as friends, winter, greet, Person as P}
```

If there is a lot to import, you can import all of them as an object:

```
import * as t from './testLib.js'

console.log(t.Person) // [class Person]
console.log(t.greet) // [Function: greet]
console.log(t.people) // [ 'Adnan Babakan', 'Arian Amini', 'Erfan Alizad', 'Alire
za Porkar' ]
console.log(t.winter) // [ 'January, February', 'March' ]
```

Default export

While exporting, you can define a default export which doesn't take a name:

```
export default class {
    constructor(firstName, lastName, age) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    introduce() {
        return `Hi! My name is ${this.firstName} ${this.lastName} adn I'm ${this.
age} years old.`
    }
}
```

Importing a default export doesn't require a specific name and can be imported by any names:

```
import Person from './Person.js'

let me = new Person('Jane', 'Doe', 30)
```

```
console.log(me.introduce()) // Hi! My name is Jane Doe and I'm 30 years old.
```
Or:

```
import P from './Person.js'

let me = new P('Jane', 'Doe', 30)
console.log(me.introduce()) // Hi! My name is Jane Doe and I'm 30 years old.
```

Exercise: Arithmetic Expression Evaluation with Dijkstra’s Two-Stack Algorithm

Algorithms are a study field in computer science, and programmers should learn them in order to be able to design fast and efficient ways to solve certain problems; and here in the section, we’ll try to learn one. Dijkstra’s two-stack algorithm is a very famous and well-known algorithm used for evaluating arithmetic expressions.

Dijkstra’s two-stack algorithm uses two stacks to store operators and values and finally returns the last evaluated value. Imagine this expression as the input: **((5 * 2) + ((9 * 3) + 6))**. The proper answer for this expression is 112, and Dijkstra’s algorithm will help us to obtain this answer using a computer program. The expressions in this exercise are fully parenthesized, and the expression is interpreted one character by one character.

Here is how Dijkstra’s two-stack algorithm works:

1. Ignore left parenthesis
2. Add numbers to the values stack
3. Add operators to ops stack
4. If encountered a right parenthesis, pop an operator and apply it on two of the operands (values).
5. Finally, return the last value left in the stack

Given the expression **((5 * 2) + ((9 * 3) + 6))**, here is how the procedure will flow:

- Push 5 to the values’ stack
- Push * to the operators’ stack
- Push 2 to the values’ stack

- Encountered “)”:
    - The last operator is *
    - The first operand is 2
    - The second operand is 5
    - Push the result (10) to the values’ stack
- Push + to the operators’ stack
- Push 9 to the values’ stack
- Push * to the operators’ stack
- Push 3 to the values’ stack
- Encountered “)”:
    - The last operator is *
    - The first operand is 3
    - The second operand is 9
    - Push the result (27) to the values’ stack
- Push + to the operators’ stack
- Push 6 to the values’ stack
- Encountered “)”:
    - The last operator is +
    - The first operand is 6
    - The second operand is 27
    - Push the result (33) to the values’ stack
- Encountered “)”:
    - The last operator is +
    - The first operator is 33
    - The second operator is 10
    - Push the result (43) the values’ stack
- The final result is 43

Here is the stack implementation we are using in this exercise:

```
class Stack {
  #items = []
  push(item) {
    this.#items.push(item)
  }

  pop() {
    if(this.#items.length < 1) return 'UNDERFLOW'
    return this.#items.pop()
  }

  top() {
    if(this.#items.length < 1) return 'NO_ITEM'
    return this.#items[this.#items.length - 1]
  }
}

module.exports = Stack
```

And now for the main part, we first require the Stack class and create a variable called input to store the expression:

```
const Stack = require('./Stack')

let input = '((5 * 2) + ((9 * 3) + 6))'
```

Then we should create two stacks, one called operators, and the other values:

```
let ops = new Stack
let vals = new Stack
```

What is required next is creating a loop to go one by one through the characters of the input and store the current character in a variable:

```
for(let i = 0; i < input.length; i++) {
    let c = input[i]
}
```

Now that we have the current character we should compare it using a switch-case statement and act accordingly:

```
for (let i = 0; i < input.length; i++) {
    let c = input[i]
    switch (c) {
        case '(':
        case ' ':
            break
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '%':
            ops.push(c)
            break
        case ')':
            break
        default:
            vals.push(parseFloat(c))
    }
}
```

So far, we checked if the character is an opening parenthesis or a space character, so we ignore it, and if it is one of the operators, it gets added to the **ops** stack. We left the actions needed to be done if an ending parenthesis is encountered blank for now, and if the character is neither of the values above, it is probably a number, so it gets added to the **vals** stack. For when an ending parenthesis is encountered, we should follow a few steps:

1. Get the last operator
2. Get the last value
3. Get the second-last value
4. According to the operator, apply the action on two operands (values)
5. Push the new value to the values' stack

And here is the final code:

```
const Stack = require('./Stack')

let input = '((5 * 2) + ((9 * 3) + 6))'

let ops = new Stack
let vals = new Stack

for (let i = 0; i < input.length; i++) {
    let c = input[i]
    switch (c) {
        case '(':
```

```
        case ' ':
            break
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '%':
            ops.push(c)
            break
        case ')':
            let op = ops.pop()
            let v = parseFloat(vals.pop())
            let vt = parseFloat(vals.pop())
            switch (op) {
                case '+':
                    v = vt + v
                    break
                case '-':
                    v = vt - v
                    break
                case '*':
                    v = vt * v
                    break
                case '/':
                    v = vt / v
                    break
                case '^':
                    v = vt ** v
                    break
                case '%':
                    v = vt % v
                    break
            }
            vals.push(v)
            break
        default:
            vals.push(parseFloat(c))
    }
}
```

Now the last value remaining in the **vals** stack is the answer we wanted. So logging it will show the answer:

```
console.log(vals.pop())
```

Chapter 9: Working with bits

JavaScript gives the developer the power to work with bits. There are several operators and methods which will help you in this case.

Binary base

Here you'll understand what binary representation is and how numbers are working in computers (everything in the computer is eventually a binary value, but here we will discuss about numbers). Turning a binary value to a decimal number (the standard human-readable number like 2, 3, -1, 912) is relatively easy. A binary value has only two digits, which are 0 and 1. Using these digits, you can represent any number as below:

Digits	1	0	1	1	1	0	0	1
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Every digit in the binary row gets multiplied by its value and finally gets summed up. In this case, our number will be as below:

$$(1 \times 2^0) + (0 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) + (1 \times 2^4) + (1 \times 2^5) + (0 \times 2^6) + (1 \times 2^7) = 185$$

So, the binary value 10111001 is equal to 185 in the decimal base. This number is called an 8-bit number since it consists of 8 bits. If all the bits of an 8-bit integer is equal to 1, it makes the number 255, which is equal to 2^8-1. So the number 255 is the maximum number we can fit in 8-bits. JavaScript uses a double-precision 64-bit binary format to handle numbers. In this format, there are 52 bits (0 to 51) dedicated to the value. The next 11 bits (52 to 62) are dedicated to the exponent. And the last bit (the 63rd bit) is the bit representing the sign (either positive or negative).

Converting a decimal number to a binary one is easy as well. What we have to do is divide the number by 2 continually (which results in a reminder of either 0 or 1) until the final result is equal to zero. For instance, let's convert the number 151 to the binary base:

Number	Result (divided-by-two)	Remainder
151	75	1
75	37	1
37	18	1
18	9	0
9	4	1
4	2	0
2	1	0

Now starting from the last result, then to the remainders, we can write it down, which in this case will be 10010111. The 10010111 number is the binary form of the decimal number 151.

Since ES6, you can use the **0b** prefix to write a number in its binary form:

```
let x = 0b10110001
console.log(x) // 177
```

And for converting a decimal number to binary (string representation of the binary form), you can use the **toString** method with the radix set to 2:

```
let x = 4
console.log(x.toString(2)) // 100
```

Bitwise operators

Bitwise operators are a part of JavaScript, which will manipulate the bits of a number.

Operator	Name	Description
&	AND	Results in 1 if both bits are 1
|	OR	Results in 1 if at least one of the bits is 1
^	XOR	Results in 1 if exactly one of the bits is 1
~	NOT	Inverts the bit (0 to 1 and 1 to 0)
<<	Zero fill left shift	Shifts the bits to the left with pushing 0 (bit) to the right and letting the leftmost bits fall off
>>	Signed right shift	Shifts the bits to the right with copying the leftmost bits and pushing them to the left and letting the rightmost bits fall off
>>>	Zero fill right shift	Shifts the bits to the right with pushing 0 (bit) to the left and letting the rightmost bits fall off

Reading all these might be confusing, but by trying them, it will be easier to understand.

AND

X	Y	Operation	Result
0	0	0 & 0	0
0	1	0 & 1	0
1	0	1 & 0	0
1	1	1 & 1	1

The numbers 6 and 3 are equal to 110 and 11 in the binary base, respectively.

```
let x = 6 & 3 // 110 & 011
```

If both bits are equal to 1 then the resulting bit will be 1, so this operator will generate the binary number 010, which is equal to 2.

```
let x = 6 & 3 // 110 & 011
console.log(x) // 2
```

OR

X	Y	Operation	Result
0	0	0 | 0	0
0	1	0 | 1	1
1	0	1 | 0	1
1	1	1 | 1	1

The numbers 13 and 10 are equal to 1101 and 1010 in the binary base, respectively.

```
let x = 13 | 10 // 1101 & 1010
```

If at least one of the bits is equal to 1, then the resulting bit will be 1, so this operator will generate the binary number 1111, which is equal to 15.

```
let x = 13 | 10 // 1101 & 1010
console.log(x) // 15
```

XOR

Sample table:

X	Y	Operation	Result
0	0	0 ^ 0	0
0	1	0 ^ 1	1
1	0	1 ^ 0	1
1	1	1 ^ 1	0

The numbers 11 and 6 are equal to 1011 and 110 in the binary base, respectively.

```
let x = 11 ^ 6 // 1011 ^ 0110
```

If exactly one bit is equal to 1, the resulting bit will be 1. So this operator will generate the binary number 1101, which is equal to 13.

```
let x = 11 ^ 6 // 1011 ^ 0110
console.log(x) // 13
```

NOT

Sample table:

X	Operation	Result
0	~0	1
1	~1	0

The number 27 is equal to 11011 in the binary base. What is essential here is to pay attention to how JavaScript stores numbers, which is the double-precision 64-bit binary format. But all bitwise operations are performed on 32-bit signed integers. So before a bitwise operation is performed, JavaScript turns the number to a 32-bit signed integer, and after the operation is done, it will turn it back to the double-precision 64-bit. The leftmost bit in a 32-bit format indicates the sign of the number. If it is 0, then the integer is positive, and if it is 0, the integer is negative. So the full form of number 27 in 32-bit signed format is as below:

00000000000000000000000000011011

And after inverting its bits, it will be as:

11111111111111111111111111100100

So ~27 won’t be equal to 4, but rather it will be equal to -28:

```
let x = ~ 27
console.log(x) // -28
```

Zero fill left shift

This operator will push the bits to the left while adding 0 bits to the right and let the leftmost bits fall off (if there are more than 32 eventually).

Number	Decimal	Operation	Result
10	1010	10 << 1	10100
52	110100	52 << 3	110100000

```
let x = 52 << 3
console.log(x) // 416
```

Signed right shift

This shift will copy the left most bit and push it from the left and let the rightmost bit fall off. This will preserve the sign of the integer, either being positive or negative.

Number	Binary	Result
9	00000000000000000000000000001001	9
9 >> 2	00000000000000000000000000000010	2
-8	11111111111111111111111111111000	-8
-8 >> 5	11111111111111111111111111111111	-1

```
let x = 9 >> 2
console.log(x) // 2
let y = -8 >> 5
console.log(y) // -1
```

Zero fill right shift

This operator will push 0 bits from the left and let the rightmost bit fall off. This operator won't preserve the sign of the integer, and the resulting integer will always be positive if at least one shift is performed.

Number	Binary	Result
12	00000000000000000000000000001100	12
12 >>> 3	00000000000000000000000000000001	1
-15	11111111111111111111111111110001	-15
-15 >>> 2	00111111111111111111111111111100	1073741820

```
let x = 12 >>> 3
console.log(x)
let y = -15 >>> 2
console.log(y)
```

Chapter 10: Asynchronous Programming

JavaScript is a single-threaded programming language, which means that JavaScript runs your code from top to bottom, and it can only execute one statement at a time. If JavaScript can execute only one statement at a time, how can JavaScript handle a program without blocking and taking so long to respond? Asynchronous programming is the answer.

What is asynchronous programming?

Asynchronous programming is a way to execute blocks of code without making others wait for it to finish. Here is how to define synchronous programming and asynchronous programming in both multi-thread and single-thread programming: Imagine you want to fill two glasses of water; what you'd do is one of three options:

1- Synchronous: You put the first glass and open the tap, then wait for it to fill. After the first glass is full, you'll do the same for the second glass.

2- Asynchronous, single-threaded: You put the first glass and then open the tap. While the first glass is being filled, you put the second glass under another tap and start to fill it as well. Then you wait for each of them to finish (the second glass can finish sooner if it has less space) and grab them.

3- Asynchronous, multi-threaded: You put both the glass at the same time under two taps and start filling them.

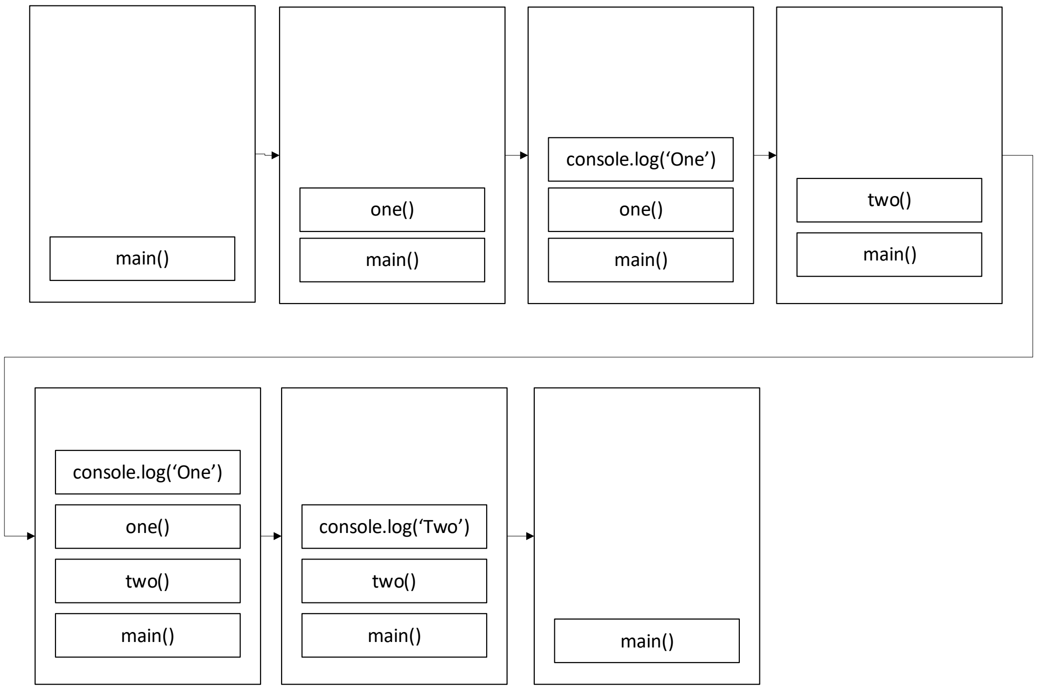
As such, JavaScript treats an asynchronous program the second way mentioned above. JavaScript uses a call stack (often referred to as execution stack), which holds what is needed to be executed. To understand this matter, let's follow this simple program's execution:

```
function one() {
    console.log('One')
}

function two() {
    one()
    console.log('Two')
}

one()
two()
```

This program's output is as below:

```
One

One

Two
```

Here is how the call stack works:



The **main()** is a wrapper used by Node.js and stays in the call stack until the program finishes. Each function adds what it needs to run, and after doing so, it gets out of the stack. This was a simple program, but what if we used something as **setTimeout**?

setTimeout is a function that delays a specific task. And here is its signature:

setTimeout(function, delay)

```
setTimeout(() => console.log("Hello World"), 1000)
```

If you run this code, you'll see the "Hellow World" string being printed after 1 second.

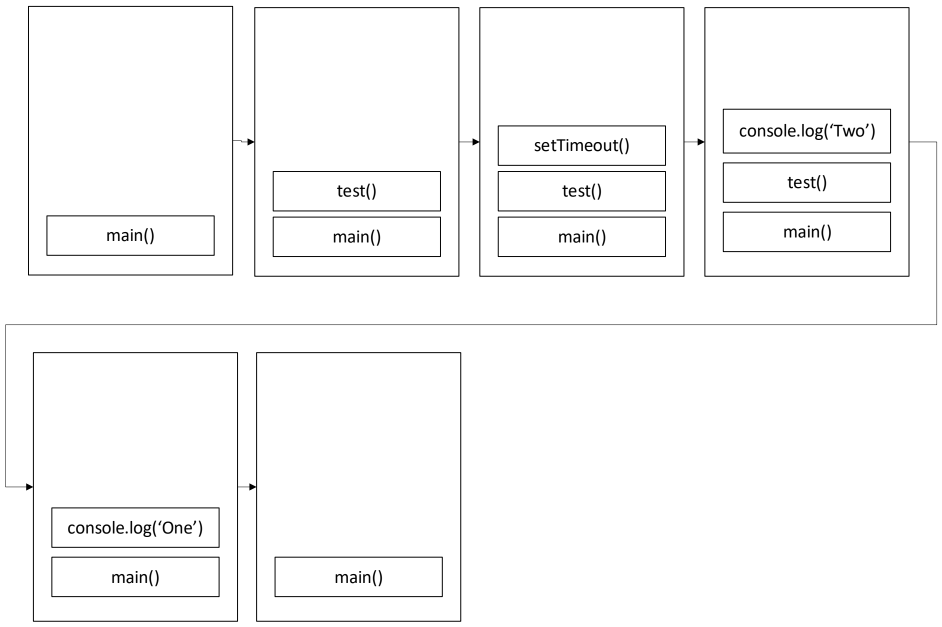
Now that we now about **setTimeout**, let's look at this code:

```
function test() {
    setTimeout(() => console.log('One'), 0)
    console.log('Two')
}

test()
```

Although **setTimeout** has 0 milliseconds of delay, the output will be as below:

```
Two

One
```

Here is how the call stack looks:



As you can, **setTimeout** appears and disappears immediately. That's because it gets added to the callback queue. The callback queue is where Node.js holds callback functions. Then event loop checks if the stack is empty and adds what is needed to be executed to the call stack, and then the procedure continues. This is what gives JavaScript the power to perform asynchronous tasks.

Callbacks

Callbacks are a huge part of JavaScript and can be quite tricky sometimes, but still really easy to understand if you listen to them! Callbacks are functions (can be both normal functions or arrow functions) that are invoked in a specific situation, usually when another function finished its job or at a certain point to perform an action. Callbacks are usually used when we are not sure how long it takes to perform a specific task such as reading.

Callback functions are passed to another function as an argument to perform a particular task, usually being an asynchronous one. Although callbacks can be used to perform a synchronous action, this usually seems redundant hence it is prevented as a practice.

To get a closer look at a callback function, have a look at the code below:

```
function sum(x, y) {
    console.log(x + y)
}

function subtract(x, y) {
    console.log(x - y)
}

function doMath(x, y, callback) {
    callback(x, y)
}

doMath(10, 8, sum) // 18
doMath(8, 3, subtract) // 5
```

doMath function gets a callback, which is a function and invokes it with the parameters passed to it. Once we passed the **sum** function and the other time, we passed the **subtract** function, which we got different results, respectively.

Caution: You should not pass your callback including the parenthesis. If you do so, it means you are getting the result of that function, which is not a function probably. So you need to pass the name of the function instead of invoking it. Which makes the code below invalid in our case:

```
doMath(10, 8, sum())
doMath(8, 3, subtract())
```

You can also define an anonymous function as your callback:

```
function doMath(x, y, callback) {
    callback(x, y)
}

doMath(5, 2, function(a, b) {
    console.log(a + b)
})
```

In the example above, our callback is executed immediately, which makes it synchronous. Most of the callbacks are used in asynchronous actions, as mentioned before. Here we will try some built-in modules from Node.js, which they use callbacks.

Working with files is a common task in programming, especially when learning to program. There is a built-in module for Node.js called **fs** (stands for File System) that helps you with this matter, providing you with methods to read, write files. (File system is briefly explained here and will be discussed more in later chapters.)

Just like we learned in the previous chapter, we should first require our module and assign it to a constant or variable.

```
const fs = require('fs')
```

Now by using the **fs** constant, we can have access to its methods. Here we will try to read and write a text file and practice the callbacks concept.

To read a file, there is a method called **readFile** we can use, and here is its signature:

fs.readFile(path[, options], callback)

The callback itself takes two arguments called. The first one holds the error, and the second one holding the file data that has been read.

For instance, in the example below, we try to read the **awesome.txt** file and print its content:

```
const fs = require('fs')

fs.readFile('awesome.txt', function(err, data) {
    console.log(data)
})
```

But if you run the code below you'll see a result like this:

```
<Buffer 48 65 79 20 74 68 69 73 20 69 73 20 61 77 65 73 6f 6d 65 21>
```

This is the buffer data of the file. If we wanted to get the text which follows an encoding, it is possible to pass a second argument to the **readFile** method identifying the encoding we want:

```
const fs = require('fs')

fs.readFile('awesome.txt', 'UTF-8', function(err, data) {
    console.log(data)
})
```

Now you can see the real text in the output.

To get a better understanding of the importance of callbacks, you can run the code below:

```
const fs = require('fs')

let fileData;

fs.readFile('awesome.txt', 'UTF-8', function(err, data) {
    fileData = data
})

console.log(fileData)
```

Here we assigned our **data** to a global variable called **fileData** and logged it outside of the callback function. But the result is going to be undefined since our **console.log** runs before our file is read. This is an example of the asynchronousness of JavaScript. A callback is called only when the task is done, which might take few milliseconds or more than 10 seconds. Combining **setTimeout** with the previous example, you can log the **fileData** variable after 1 second, which you might get the proper data:

```
const fs = require('fs')
```

```
let fileData;

fs.readFile('awesome.txt', 'UTF-8', function(err, data) {
    fileData = data
})

console.log(fileData) // undefined

setTimeout(() => console.log(fileData), 1000) // Hey this is awesome!
```

Caution: This is not the valid way of using your data out of callback and only mentioned here in a matter of understanding that callbacks are called when they are needed and not immediately.

Another method available in **fs** module is **writeFile,** which enables you to write content to a file. And here is its signature:

fs.writeFile(file, data[, options], callback)

The callback here is used to determine if there are any errors and has only one argument holding the error.

Let's write a text into a file called **me.txt**:

```
const fs = require('fs')

fs.writeFile('me.txt', 'My name is Adnan Babakan', function(e) {
    if(e) console.log(e)
})
```

Promises

Promises represent the completion or failure of an asynchronous task. Promises are easy to use and a relatively new feature of JavaScript.

Here is how MDN defines promises:

> *A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the*

final value, the asynchronous method returns a promise to supply the value at some point in the future.

A promise has three states:

- pending: This is the initial state of every promise
- fulfilled: This is the state of a promise when the operation is completed successfully
- rejected: This is the state of a promise when the operation has failed

A promise is an object that can be created using the **new** keyword and **Promise** constructor. The first argument passed is a function that takes two arguments, **resolve** and **reject**. When resolve is called, it means the promise has been fulfilled, on the other hand, if reject is called, it means the promise has been rejected.

Here is how to create a promise:

```
let myPromise = new Promise((resolve, reject) => {

})
```

Each promise can be handled with three chained blocks:

- then: This block is run when the promise is fulfilled
- catch: This block is run when the promise is rejected
- finally: This method is run eventually after **then** or **catch** no matter what the status of the promise is

To test **myPromise** we can resolve the promise using a **setTimeout**:

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Done'), 3000)
})

myPromise.then(value => console.log(value))
```

As you can see, resolve is taking one argument here, so our **then** method will also take a function with one argument to handle them, respectively. If you run the code above, you should see “Done” being printed in your command line after 3 seconds.

Now, if we have rejected this promise, this is how we should have handled the error:

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => reject('Not so good!'), 3000)
})

myPromise.catch(err => console.log(err))
```

Let’s write a simple function to return a promise and resolve or reject it depending on the argument passed to the function:

```
let isLegalAge = age => {
    return new Promise((resolve, reject) => {
        if(age > 18) {
            resolve('Yes')
        } else {
            reject('No')
        }
    })
}

isLegalAge(10)
    .then(v => console.log(v))
    .catch(e => console.log(e))
```

As you can see I called my function with an argument and handled my promise according to that value being more or less than 18. You can use the **finally** method as well:

```
isLegalAge(10)
    .then(v => console.log(v))
    .catch(e => console.log(e))
    .finally(() => console.log('Task done!'))
```

No matter if the promise resolves or rejects, the function passed to the **finally** method will be run. So the result of the code above will be like below:

```
No

Task done!
```

All the tasks done above are pretty much immediately done if we don’t consider the **setTimeout,** and using a promise is not practical and feels useless. But in the case of working with asynchronous methods using a promise can make your code much more concise and understandable and can help you in many situations. For instance, here is a promise that resolves a file data and rejects with an error if reading the file encounters an error:

```
const fs = require('fs')

let readFilePromise = file => {
    return new Promise((resolve, reject) => {
        fs.readFile(file, 'UTF-8', (err, data) => {
            if(err) {
                reject(err)
            } else {
                resolve(data)
            }
        })
    })
}

readFilePromise('awesome.txt')
    .then(d => console.log(d))
    .catch(e => console.log(e))
```

Promises are usuefull especially when combining multiple callbacks such as when making an HTTP server and serving files inside which will be covered in the later chapters.

Chained promises

When working with multiple asynchronous operations, we can chain the promises. Returning anything in the **then** block will make you able to chain another **then** block. Here is how:

```
let multiply = (x, y) => {
    return new Promise((resolve, reject) => {
        resolve(x * y)
    })
}

multiply(9, 8)
    .then(v => {
        console.log(v)
        return v * 5
    })
    .then(v => {
        console.log(v)
    })
```

As you can see, I’ve returned a number in my first, **then** that makes it possible to receive in the second **then** block.

A **then** block can return a new promise object as well:

```
let multiply = (x, y) => {
    return new Promise((resolve, reject) => {
        resolve(x * y)
    })
}

multiply(4, 4)
    .then(v => {
        console.log(v)
        return new Promise((resolve, reject) => resolve(v ** 2))
    })
    .then(v => {
        console.log(v)
    })
```

The result of the code above will be:

```
16

256
```

If a **then** block returns a promise that rejects, the error will be received by the catch block chained to the main chain:

```
multiply(4, 4)
    .then(v => {
        console.log(v)
        return new Promise((resolve, reject) => reject("Error"))
    })
    .then(v => {
        console.log(v)
    })
    .catch(e => console.log(e))
```

Meaning that the **catch** block is shared between the main promise and the promise returned from the first **then** block. Alternatively, you can return the promise itself in a **then** block:

```
let cube = n => {
    return new Promise((resolve, reject) => {
        resolve(n ** 3)
    })
}
```

```
cube(2)
    .then(v => {
        console.log(v)
        return cube(v)
    })
    .then(v => {
        console.log(v)
        return cube(v)
    })
    .then(v => {
        console.log(v)
        return cube(v)
    })
```

The output of this code is as below:

```
8

16

134217728
```

Async/Await

As you can imagine using multiple promises inside each other can be a real mess. An async function can permit you to use the **await** keyword. The **await** keyword allows you to simplify the usage of a promise and wait till it resolves. An async function is defined using the **async** keyword as below:

```
async function myFirstAsyncFunction() {
    //
}
```

Here is a sample of a promise being handled inside an async function:

```
const fs = require('fs')

let readFilePromise = file => {
    return new Promise((resolve, reject) => {
        fs.readFile(file, 'UTF-8', (err, data) => {
            if(err) reject(err)
            else resolve(data)
        })
    })
}

async function myFunc(file) {
    let data = await readFilePromise(file)
    console.log(data)
}

myFunc('awesome2.txt')
```

As you see, there is no **then** block, and we only used our promise with an **await** keyword prepended. In this situation, if our promise rejects, we can handle it using a **try/catch** block. (You’ll read about this in the next chapter)

```
async function myFunc(file) {
    try {
        let data = await readFilePromise(file)
        console.log(data)
    } catch(e) {
        console.log(e)
    }
}

myFunc('awesome2.txt')
```

It is useful to use an async function when combining multiple promises. Here we will read a file and write its content to another one using two promises and an async function:

```
const fs = require('fs')

let readFilePromise = file => {
    return new Promise((resolve, reject) => {
        fs.readFile(file, 'UTF-8', (err, data) => {
            if(err) reject(err)
            else resolve(data)
        })
    })
}

let writeToFilePromise = (file, data) => {
    return new Promise((resolve, reject) => {
        fs.writeFile(file, data, (err) => {
            if(err) reject(err)
            else resolve('Done')
        })
    })
}

async function copy(fileToRead, fileToWrite) {
    try {
        let data = await readFilePromise(fileToRead)
        await writeToFilePromise(fileToWrite, data)
    } catch(e) {
        console.log(e)
    }
}

copy('awesome.txt', 'awesome2.txt')
```

This way it is cleaner and readable to use multiple promises and manage their resolves and rejects, but it is still your choice to use either way.

Chapter 11: Errors and Error Handling

In every program by any programmer, there might be several errors, and there is no shame in it but not understanding them or not handling them is shameful. Node.js is no exception and can experience four categories of errors:

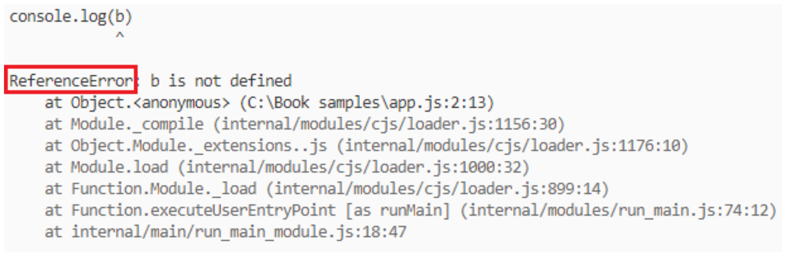
1- Standard JavaScript errors
   a. <EvalError>
   b. <SyntaxError>
   c. <RangeError>
   d. <ReferenceError>
   e. <TypeError>
   f. <URIError>
2- System errors
3- User-specified errors
4- AssertionError

All JavaScript errors and System errors raised by Node.js inherit from or are instances of, the standard JavaScript **<Error>** class.

The type of error can be determined from the error log. The code below throws an error of type **ReferenceError**.

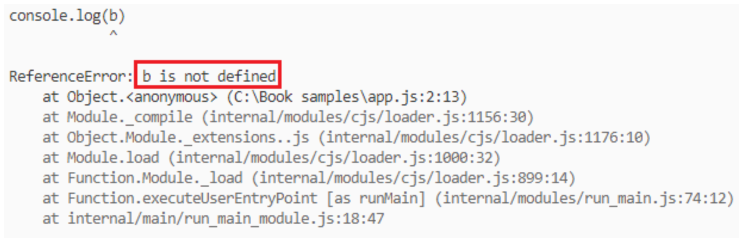
```
let a = 5
console.log(b)
```

And is seen in the log:



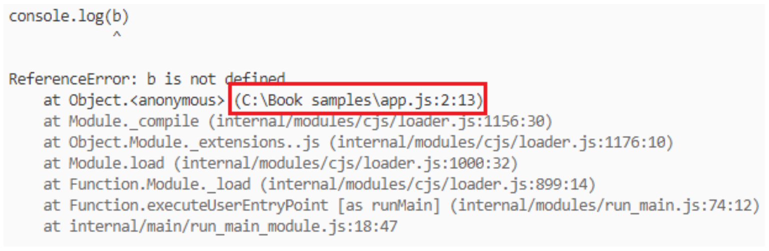
```
console.log(b)
            ^

ReferenceError: b is not defined
    at Object.<anonymous> (C:\Book samples\app.js:2:13)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47
```

Each error usually has some description of what is happening in front of the error type:



```
console.log(b)
            ^

ReferenceError: b is not defined
    at Object.<anonymous> (C:\Book samples\app.js:2:13)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47
```

As well it shows you where the error is happening:



```
console.log(b)
            ^

ReferenceError: b is not defined
    at Object.<anonymous> (C:\Book samples\app.js:2:13)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47
```

“C:\Book samples\app.js:2:13” indicates that there is an error in a file located at **C:\Book samples\app.js** on line 2, column 13.

It is really important and crucial to know how to read an error since it will help you move forward faster if there are any errors.

Throw and Try/Catch

Errors are usually thrown, which should be caught. If not, Node.js will exit the program immediately. Throwing an error is done by the **throw** keyword, and for handling it, there is a statement called **try/catch**. Throwing errors is usually done by the programmer specifically or by most of the synchronous APIs (You will read about synchronous and asynchronous APIs later).

An error can be thrown like below:

```
throw 'This is my error!'
```

And the output of the program would be as this:

```
throw "This is my error!"

^

This is my error!

(Use `node --trace-uncaught ...` to show where the exception was thrown)
```

Any code after a thrown error in case it is not handled won't be executed since the program gets quit as soon as an error is thrown. So the code below will only print the first line, and then after throwing an error, there will be no results.

```
console.log('First line')
throw 'I\'m an error!'
console.log('Second line')
```

By using the **try/catch** statement, we can prevent such incidents from happening. Any error thrown inside a **try** block will immediately be handled by a **catch** block receiving the error.

```
try {
    throw 'Kaboom!'
} catch(error) {
    console.log(error)
}
```

Now that an error is thrown inside a **try/catch**, it won't bother the rest of the program anymore, and if there are any codes after it, they can be executed normally.

Having the code below:

```
console.log('First log')

try {
    throw 'Kaboom!'
} catch(error) {
    console.log(error)
}

console.log('Second log')
```

The result in the command line would be as below:

```
First log
```

```
Kaboom!

Second log
```

It might seem obsolete to throw and catch an error like this, and yes, it is! But sometimes it is not you who throws an error. It can be some other API, some other function, or anything else. For instance, when reading a file using the file system (will be discussed more later), if a file is not present, it can throw an error:

```
const fs = require('fs')

fs.readFileSync('./non_existing_file.txt')
```

This code generates the output below:

```
internal/fs/utils.js:230

    throw err;

    ^



Error: ENOENT: no such file or directory, open './non_existing_file.txt'

    at Object.openSync (fs.js:458:3)

    at Object.readFileSync (fs.js:360:35)

    at Object.<anonymous> (C:\Book samples\app.js:3:4)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47 {

  errno: -4058,
  syscall: 'open',

  code: 'ENOENT',

  path: './non_existing_file.txt'

}
```

Which stops the program as we've seen in the previous example. But it can be handled so there won't be any problem:

```
const fs = require('fs')

try {
    fs.readFileSync('./non_existing_file.txt')
} catch(e) {
    console.log(e)
}
```

Caution: A try/catch statement has its own scope.

Error class

JavaScript has a generic **Error** class that defines an error regardless of the type, and other error types extend this class. The constructor of this class receives a message.

```
throw new Error('I am an error!')
```

This code generates the error logged below:

```
throw new Error('I am an error!')

^



Error: I am an error!

    at Object.<anonymous> (C:\Book samples\app.js:1:7)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

This error can also be handled by a **try/catch** statement:

```
try {
    throw new Error('I am an error!')
} catch(e) {
    console.log(e.name) // Error
    console.log(e.message) // I am an error!
}
```

<EvalError>

EvalError is a kind of error that is thrown regarding the **eval** function. **Eval** function executes the string inside it as a JavaScript code. Using the **Eval** function is discouraged in almost 99% of the cases.

```
eval('let a = 5; console.log(a)') // 5
```

if there is an error while evaluating the string, an error of <EvalError> type would be thrown usually, but this error type is no longer used by JavaScript. Although this error type is not thrown by JavaScript anymore, the **EvalError** object remains fully functional for compatibility and can be thrown manually:

```
try {
    throw new EvalError('I am an EvalError!')
} catch (e) {
    console.log(e.name) // EvalError
    console.log(e.message) // I am an EvalError!
}
```

<SyntaxError>

SyntaxError is a really common error. A **SyntaxError** is thrown if there is any mistake regarding the interpretation of code, such as a typo in which JavaScript cannot determine what keyword you are trying to use.

```
console@log('Hello World')
```

In the code above, we've replaced **"."** with **"@"** which has no meaning thus, JavaScript won't be able to interpret this code and throws a Syntax Error.

```
console@log('Hello World')

       ^



SyntaxError: Invalid or unexpected token

    at wrapSafe (internal/modules/cjs/loader.js:1070:16)

    at Module._compile (internal/modules/cjs/loader.js:1120:27)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

Caution: A SyntaxError cannot be handled since JavaScript would immediately terminate the program if it cannot understand what the program syntax is following.

```
try {
    console@log('Hello')
} catch(e) {
    console.log(e)
}

console.log('Here')
```

As expected, this code won't work, and the last line won't be executed. But if a Syntax Error is thrown manually and by using the **SyntaxError** object, it can be caught:

```
try {
    throw new SyntaxError('I am a SyntaxError')
} catch (e) {
    console.log(e.name) // SyntaxError
    console.log(e.message) // I am a SyntaxError
}

console.log('Here')
```

<RangeError>

RangeError occurs when trying to pass a number not allowed. Such as when trying to pass a bad value to **Number.prototype.toExponential()**. This function returns a string representation of the number in exponential notation.

```
console.log(Number.parseFloat(1292).toExponential(2)) // "1.29e+3"
```

The number 2 passed is the value showing how many digits should there be after the decimal point. This argument should be from 0 to 100 (inclusive), and if it is more or less than them, it will throw a **RangeError**. So this code:

```
console.log(Number.parseFloat(1292).toExponential(101))
```

Results in this error:

```
console.log(Number.parseFloat(1292).toExponential(101))
                                    ^



RangeError: toExponential() argument must be between 0 and 100

    at Number.toExponential (<anonymous>)

    at Object.<anonymous> (C:\Book samples\app.js:1:37)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

By using the **RangeError** object constructor we can throw this kind of error manually:

```
try {
    throw new RangeError('Hi I am a RangeError!')
} catch (e) {
    console.log(e.name) // RangeError
    console.log(e.message) // Hi I am a RangeError!
}
```

<ReferenceError>

ReferenceError is another common error that occurs when trying to access something in JavaScript, which is not defined. Such as when trying to access a function or a variable that is not defined:

```
console.log(y)
```

Or

```
myFunction()
```

This code throws an error as below:

```
myFunction()

^



ReferenceError: myFunction is not defined

    at Object.<anonymous> (C:\Book samples\app.js:1:1)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

And here is the result of handling:

```
try {
    myFunction()
} catch (e) {
    console.log(e.name) // ReferenceError
    console.log(e.message) // myFunction is not defined
}
```

<TypeError>

TypeError happens when an operation cannot be performed. Such as when a value is not of the expected type. This kind of error can happen if you are trying to access a property of a null:

```
null.f()
```

This code will generate the error below:

```
null.f()

     ^



TypeError: Cannot read property 'f' of null

    at Object.<anonymous> (C:\Book samples\app.js:1:6)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

And here is the logs of handling this kind of error:

```
try {
    null.f()
} catch (e) {
    console.log(e.name) // TypeError
    console.log(e.message) // Cannot read property 'f' of null
}
```

<URIError>

URIError is a rare type of error and is only occurred when trying to work with a URI (Uniform Resource Identifier). This is how Wikipedia defines URI:

A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource.

Simply URI is the address you are using to access a specific resource such as when trying to access a specific page on the web:

http://example.com/category/post?id=5

A URI can be encoded by using the **encodeURI()** function to form a UTF-8 string:

```
console.log(encodeURI('http://example.com/seacrh=This is a query')) // http://exa
mple.com/seacrh=This%20is%20a%20query
```

And can be decoded by using the **decodeURI()** function:

```
console.log(decodeURI('http://example.com/seacrh=This%20is%20a%20query')) // http
://example.com/seacrh=This is a query
```

If the value passed to this function is not of a valid URI, a URIError will be thrown.

```
decodeURI('%')
```

The code above generates the error below:

```
decodeURI('%')

^



URIError: URI malformed

    at decodeURI (<anonymous>)
    at Object.<anonymous> (C:\Book samples\app.js:1:1)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

And here is the logs of handling this kind of error:

```
try {
    decodeURI('%')
} catch (e) {
    console.log(e.name) // URIError
    console.log(e.message) // URI malformed
}
```

System errors

System errors occur when exceptions occur within Node.js's runtime. These errors usually happen when an operating system constraint is violated. A system error can have these properties if they are available:

1- address: The address to which a network connection failed
2- code: The error code
3- dest: The file path destination when reporting a file system error
4- errno: The system-provided error number
5- info: Extra details about the error
6- message: A human-readable description of the error
7- path: The file path when reporting a file system error
8- port: The network connection port that is not available
9- syscall: The name of the system call that triggered the error

Here are a few of the system errors:

1- EACCES: This error happens when an attempt was made to access a file forbidden by the operating system.
2- EADDRINUSE: This error happens when trying to bind an address to the server while it's already being used.
3- ENOENT: This error happens when trying to access a file or directory that does not exist.

User-specified errors

When designing an application, it is sometimes needed to have custom errors which only have meaning in this specific application. A user-specified error can be as simple as throwing an error provided only with a string. But it is a good practice to define a custom error using a class extending JavaScript's **Error** class.

```
class MyError extends Error {

}
```

A custom error object needs to have a property called **message,** which is going to be shown in the error log. Since a custom error class extends the default **Error** class, it is needed to pass the constructor arguments to the parent class's constructor. Thus we need to use the **super** keyword.

```
class MyError extends Error {
    constructor(message) {
        super(message)
    }
}
```

So far our custom error class is done. Its message gets set by the constructor of the **Error** class so no further action is needed. Now it is possible to export this class and throw an error:

```
class MyError extends Error {
    constructor(message) {
        super(message)
    }
}

module.exports = MyError
```

And here is the code to throw an error using this class:

```
const MyError = require('./MyError')
```

```
throw new MyError('It is brand new!')
```

The code above generates the output below:

```
throw new MyError('It is brand new!')

^



MyError: It is brand new!

    at Object.<anonymous> (C:\Book samples\app.js:3:7)

    at Module._compile (internal/modules/cjs/loader.js:1156:30)

    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)

    at Module.load (internal/modules/cjs/loader.js:1000:32)

    at Function.Module._load (internal/modules/cjs/loader.js:899:14)

    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)

    at internal/main/run_main_module.js:18:47
```

If a class doesn't extend the **Error** class it cannot be thrown:

```
class MyError {
    constructor(message) {
        this.message = message
    }
}

module.exports = MyError
```

If you try to throw an error using this class as below:

```
const MyError = require('./MyError')

throw new MyError('Not an error class!')
```

Here is the output Node.js will generate:

```
throw new MyError('Not an error class!')
```

```
^

MyError { message: 'Not an error class!' }
```

So it literally throws and object and not an error. Although this can terminate the program, the identity of this object is not an error thus, there are no traces and no error message that Node.js can understand.

A custom error can have any property wanted and can be accessed while catching that exception, but here are the usual ones:

1- address
2- code
3- dest
4- errno
5- info
6- message
7- path

So far, you are familiar with the message property, and the rest of the properties are pretty much self-explanatory.

Here is a custom error class with more properties:

```
class MyError extends Error {
    constructor(message) {
        super(message)

        this.code = 'MY_ERROR'
        this.info = 'You need to change something!'
    }
}

module.exports = MyError
```

Throwing an error using this class generates the output below:

```
throw new MyError('Not an error class!')

^
```

```
MyError: Not an error class!
    at Object.<anonymous> (C:\Book samples\app.js:3:7)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47 {
  code: 'MY_ERROR',
  info: 'You need to change something!'
}
```

And by catching this error it is possible to access each of the properties individually:

```
const MyError = require('./MyError')

try {
    throw new MyError('Kaboom!')
} catch(e) {
    console.log(e.message) // Kaboom
    console.log(e.code) // MY_ERROR
    console.log(e.info) // You need to change something!
}
```

It is practical to include another property called **name** which holds the error class's name. It is possible to retrieve a class's name by using **this.constructor.name**:

```
class MyError extends Error {
    constructor(message) {
        super(message)

        this.name = this.constructor.name
        this.code = 'MY_ERROR'
        this.info = 'You need to change something!'
    }
}

module.exports = MyError
```

Now it is possible to see the error name:

```
const MyError = require('./MyError')

try {
    throw new MyError('Kaboom!')
} catch(e) {
    console.log(e.message) // Kaboom
    console.log(e.name) // MyError
    console.log(e.code) // MY_ERROR
    console.log(e.info) // You need to change something!
}
```

If you don't set the name of your error class, the default name will be **Error** since it is inherited from the **Error** class.

AssertionError

AssertionError is a special kind of error thrown by assertions. An assertion is a strict way of checking the truthy of values. If the condition is not met, it will throw an AssertionError, and the program will terminate immediately. In order to use assertions, there is a built-in library called **assert,** which needs to be imported.

```
const assert = require('assert')
```

This library provides us with a way to test the truthy of value just as below:

```
assert(7 > 2)
```

This value evaluates to true so nothing will happen, but if there is a falsely value, then an AssertionError will be thrown:

```
assert(8 < 2)
```

This assert generates the error below:

assert.js:385

```
    throw err;
    ^

AssertionError [ERR_ASSERTION]: The expression evaluated to a falsy value:

  assert(8 < 2)

    at Object.<anonymous> (C:\Book samples\app.js:3:1)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47 {
  generatedMessage: true,
  code: 'ERR_ASSERTION',
  actual: false,
  expected: true,
  operator: '=='
}
```

The **assert** function can receive a custom message to be shown when the **AssertionError** is thrown:

```
assert(8 < 2, 'Hmmm... Something seems wrong!')
```

This code throws an error generating the log below:

```
assert.js:385
    throw err;
    ^


AssertionError [ERR_ASSERTION]: Hmmm... Something seems wrong!
    at Object.<anonymous> (C:\Book samples\app.js:3:1)
    at Module._compile (internal/modules/cjs/loader.js:1156:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1176:10)
    at Module.load (internal/modules/cjs/loader.js:1000:32)
    at Function.Module._load (internal/modules/cjs/loader.js:899:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:74:12)
    at internal/main/run_main_module.js:18:47 {
  generatedMessage: false,
  code: 'ERR_ASSERTION',
  actual: false,
  expected: true,
  operator: '=='
}
```

As you can see, our error message is shown in the error log.

Assert library has more than only one function which they have different usage and are really helpful. Some of them are explained below.

assert.deepStrictEqual(actual, expected[, message])

Checks the equality of the actual with the expected arguments deeply, meaning that the children properties will be checked recursively.

```
assert.deepStrictEqual({a: 7}, {b: 7})
```

This assert throws an error since the objects are not equal.

```
assert.deepStrictEqual({n: [7, 8, 9]}, {n: [7, 8, 10]})
```

This assert will throw an error as well since the items in the array are not equal.

assert.notDeepStrictEqual(actual, expected[, message])

Just as the opposite of the previous method, this methods checks for the inequality of the actual with the expected arguments.

```
assert.notDeepStrictEqual({n: [7, 8, 10]}, {n: [7, 8, 10]})
```

The code above throws an error since two arguments are deeply equal.

assert.ifError(value)

Throws an error if the value passed to the method is not undefined or null.

```
assert.ifError(null)
```

The code above won’t throw an error since the value is null, but the code below throws an error since the value passed to it is a string.

```
assert.ifError('0')
```

This method is especially useful when testing the error argument in callbacks.

Error handling in callbacks

Callback functions by convention, have an argument specifically for errors, which is places as the first argument. These kinds of errors won’t terminate the program unless specified to do so. For instance, when reading a file, if the file doesn’t exist, the first argument will include the error.; if not, it will be null.

```
const fs = require('fs')

fs.readFile('test.txt', 'UTF-8', function(err, data) {
    console.log(err)
})
```

The **err** argument in this callback is equal to the output below since the file specified is not available:

```
[Error: ENOENT: no such file or directory, open 'C:\Book samples\test.txt'] {

  errno: -4058,

  code: 'ENOENT',

  syscall: 'open',

  path: 'C:\\Book samples\\test.txt'

}
```

But if the file exists, here is how the code works:

```
const fs = require('fs')

fs.readFile('test.txt', 'UTF-8', function(err, data) {
    console.log(err) // null
})
```

Such an error can be handled using a simple if statement. A null value is a falsely value, so the block of code in an if statement won't run in case there is no error.

```
const fs = require('fs')

fs.readFile('test.txt', 'UTF-8', function(err, data) {
    if(err) {
        console.log(err)
    }
})
```

Chapter 12: Date

One of the most common libraries ever needed while working on a project is the date! JavaScript has this library built-in, and you can create an object of date as simple as this:

```
let d = new Date
console.log(d) // 2020-07-22T21:10:48.258Z
```

The letter **Z** at the end of a date/time string is indicating that this date is based on UTC.

Tip: UTC stands for Universal Time.

Constructor

A date constructor accepts a date string as the date you want to work with. If left blank, it will be considered that you need the current date.

It is possible only to pass a year:

```
let d = new Date('2010')
console.log(d) // 2010-01-01T00:00:00.000Z
```

It is possible to add a month and day (or just month):

```
let d = new Date('2010-10-03')
console.log(d) // 2010-10-03T00:00:00.000Z
```

Caution: The format of the date should always be YYYY/MM/DD.

Caution: It is necessary to add a leading zero to a one-digit number. For instance, if you want to indicate the 3rd day of a month, you should put it as 03. If you don't do so, it is highly likely to encounter unexpected behavior.

Here is how to set the hour, minute and second, and millisecond:

```
let d = new Date('2010-10-30T22:40:30.800')
console.log(d) // 2010-10-30T19:10:30.800Z
```

As you can see, the time might be a little bit off (and sometimes date can also be). This is because we are defining to timezone. If you want to set the timezone as UTC, you need to add the letter **Z** at the end:

```
let d = new Date('2010-10-30T22:40:30.800Z')
console.log(d) // 2010-10-30T22:40:30.800Z
```

For other timezones, you can use a plus or minus sign and add up or subtract the difference from UTC:

```
let d = new Date('2010-10-30T22:40:30.800+04:30')
console.log(d) // 2010-10-30T18:10:30.800Z

let d = new Date('2010-10-30T22:40:30.800-04:30')
console.log(d) // 2010-10-31T03:10:30.800Z
```

Methods

There are several methods that this object provides us with:

Getters

These methods are used for retrieving information:

getFullYear()

Returns a four-digit representation of the current year:

```
console.log(d.getFullYear()) // 2020
```

getMonth()

Returns the current month from 0 to 11:

```
console.log(d.getMonth()) // 6
```

JavaScript counts months from 0 to 11, meaning that January is 0 and December is 11.

getDate()

Returns the day of the month from 1 to 31:

```
console.log(d.getDate()) // 22
```

getDay()

Returns the day of the week from 0 to 6:

```
console.log(d.getDay()) // 3
```

getHours()

Returns the current hour from 0 to 23:

```
console.log(d.getHours()) // 19
```

getMinutes()

Returns the current minute from 0 to 59:

```
console.log(d.getMinutes()) // 38
```

getSeconds()

Returns the current second from 0 to 59:

```
console.log(d.getSeconds()) // 6
```

getMilliseconds()

Returns the current millisecond from 0 to 999:

```
console.log(d.getMilliseconds()) // 510
```

getTime()

Returns the milliseconds passed since midnight 1st of January, 1970:

```
console.log(d.getTime()) // 1595430791212
```

Tip: The date “1st of January, 1970” is called “epoch date” and is the starting date of Unix operating systems.

getTimezoneOffset()

Return the time difference between UTC and local time in minutes:

```
console.log(d.getTimezoneOffset()) // -270
```

getUTCFullYear()

Returns the year according to UTC:

```
console.log(d.getUTCFullYear()) // 2020
```

getUTCMonth()

Returns the month according to UTC:

```
console.log(d.getUTCMonth()) // 6
```

getUTCDate()

Returns the day of the month according to UTC:

```
console.log(d.getUTCDate()) // 22
```

getUTCDay()

Returns the day of the week according to UTC:

```
console.log(d.getUTCDay()) // 3
```

getUTCHours()

Returns the hour according to UTC:

```
console.log(d.getUTCHours()) // 15
```

getUTCMinutes()

Returns the minute according to UTC:

```
console.log(d.getUTCMinutes()) // 35
```

getUTCSeconds()

Returns the second according to UTC:

```
console.log(d.getUTCSeconds()) // 2
```

getUTCMilliseconds()

Returns the millisecond according to UTC:

```
console.log(d.getUTCMilliseconds()) // 105
```

now()

Returns the milliseconds since 1st of January, 1970. This method is not a method accessible via the object create but rather is a static method:

```
console.log(Date.now()) // 1595432712555
```

parse()

Parses a string representing a date and returns the milliseconds since the 1st of January, 1970. Just like the previous method, this one is also a static method:

```
console.log(Date.parse('January 20 2016')) // 1453235400000
```

UTC()

Returns the milliseconds since 1st of January, 1970 till the date specified. This method gets two required arguments of **year** and **month** and five other optional arguments: **day**, **hour**, **minute**, **second,** and **millisecond**.

```
console.log(Date.UTC(1980, 2)) // 320716800000
console.log(Date.UTC(2050, 10, 20, 7, 23, 42, 762)) // 2552541822762
```

Setters

These methods are used for setting the date properties:

setFullYear()

Sets the year of a date object (negative numbers allowed):

```
d.setFullYear(2025)
console.log(d) // 2025-07-22T21:11:46.672Z
```

```
d.setFullYear(-135)
console.log(d) // -000135-07-22T22:42:16.055Z
```

This method also accepts two other arguments being month and day to be set. The month argument accepts numbers from 0 to 11 representing January to December; other numbers will also be accepted. Such as -1, which represents the last month of the previous year, and 12 represents the first month of the next year. Day argument accepts numbers from 1 to 31, and respectively 0 represents the last day of the previous month while 32 is the first day of the next month.

```
d.setFullYear(2025, 5)
console.log(d) // 2025-06-22T21:15:22.151Z
```

If we set the month to -2, it will show the November of 2024:

```
d.setFullYear(2025, -2)
console.log(d) // 2024-11-22T22:16:02.821Z
```

And here is an example of the day argument:

```
d.setFullYear(2025, 4, 5)
console.log(d) // 2025-05-04T21:16:35.262Z
```

And a negative number for the day argument:

```
d.setFullYear(2025, 4, -3)
console.log(d) // 2025-04-26T21:16:50.865Z
```

setMonth()

Sets the month of a date object just like the month argument of the previous method:

```
d.setMonth(3)
console.log(d) // 2020-04-22T21:17:47.376Z
```

And a negative number as the month argument:

```
d.setMonth(-5)
console.log(d) // 2019-08-22T21:18:08.217Z
```

This method accepts a second argument to set the day, just like the day argument of the previous method:

```
d.setMonth(2, -2)
console.log(d) // 2020-02-26T22:19:17.660Z
```

setDate()

Sets the day of a date object just like the day argument of two previous methods:

```
d.setDate(28)
console.log(d) // 2020-07-27T21:21:14.070Z
```

setHours()

Sets the hour of a date object from number 0 to 23. Other numbers are accepted, while -1 represents the last hour of the previous day and 24 representing the first hour of the next day.

```
d.setHours(5)
console.log(d) // 2020-07-23T01:23:09.752Z
```

And a negative number:

```
d.setHours(-5)
console.log(d) // 2020-07-22T15:23:31.361Z
```

This method supports three other arguments: minute, second, and millisecond. The minute argument can be from 0 to 59, while -1 represents the last minute of the previous hour and 60 the first minute of the next hour. The second and millisecond arguments can be from 0 to 59 and from 0 to 999, respectively, while negative numbers and numbers beyond that range represent previous and next units.

```
d.setHours(5, 3, 1, 8)
console.log(d) // 2020-07-23T00:33:01.008Z

d.setHours(5, 8, 90, 8)
console.log(d) // 2020-07-23T00:39:30.008Z

d.setHours(5, 8, 90, -12)
console.log(d) // 2020-07-23T00:39:29.988Z
```

setMinutes()

Sets the minute of a date object just like the minute argument of the previous method:

```
d.setMinutes(5)
console.log(d) // 2020-07-22T20:35:29.576Z
```

This method accepts two other arguments: second and millisecond, just like the previous method.

```
d.setMinutes(5, 4)
console.log(d) // 2020-07-22T21:35:04.679Z

d.setMinutes(5, 4, -7)
console.log(d) // 2020-07-22T21:35:03.993Z
```

setSeconds()

Sets the second of a date object just like the second argument of the previous method:

```
d.setSeconds(9)
console.log(d) // 2020-07-22T21:35:09.023Z

d.setSeconds(-20)
console.log(d) // 2020-07-22T21:34:40.348Z
```

This method accepts a second argument to set the millisecond of a date object just like the millisecond date of the previous method:

```
d.setSeconds(30, -30)
console.log(d) // 2020-07-22T21:36:29.970Z
```

setMilliseconds()

Sets the millisecond of a date object:

```
d.setMilliseconds(570)
console.log(d) // 2020-07-22T21:36:56.570Z
d.setMilliseconds(-700)
console.log(d) // 2020-07-22T21:37:20.300Z
```

setTime()

Set a date to a specifc number of milliseconds after or before 1st of January, 1970:

```
d.setTime(0)
console.log(d) // 1970-01-01T00:00:00.000Z

d.setTime(81241278651)
console.log(d) // 1972-07-29T07:01:18.651Z

d.setTime(6247867815592)
console.log(d) // 2167-12-27T06:50:15.592Z

d.setTime(-1)
console.log(d) // 1969-12-31T23:59:59.999Z

d.setTime(-789129781252)
console.log(d) // 1944-12-29T13:16:58.748Z
```

setUTCFullYear()

Exactly like **setFullYear()**, but based on UTC.

```
d.setUTCFullYear(2070)
console.log(d) // 2070-07-22T21:39:27.824Z

d.setUTCFullYear(2070, 7, 8, 2)
console.log(d) // 2070-08-08T21:39:41.617Z
```

setUTCMonth()

Exactly like **setMonth()**, but based on UTC.

```
d.setUTCMonth(10)
console.log(d) // 2020-11-22T21:40:47.199Z

d.setUTCMonth(15, -4)
console.log(d) // 2021-03-27T21:41:08.630Z
```

setUTCDate()

Exactly like **setDate()**, but baed on UTC.

```
d.setUTCDate(10)
console.log(d) // 2020-07-10T21:44:31.111Z
```

setUTCHours()

Exactly like **setHours()**, but based on UTC.

```
d.setUTCHours(20)
console.log(d) // 2020-07-22T20:44:00.463Z
```

```
d.setUTCHours(20, 8, 10, -5)
console.log(d) // 2020-07-22T20:08:09.995Z
```

setUTCMinutes()

Exactly like **setMinutes()**, but based on UTC.

```
d.setUTCMinutes(40)
console.log(d) // 2020-07-22T21:40:17.271Z

d.setUTCMinutes(40, 7, -500)
console.log(d) // 2020-07-22T21:40:06.500Z
```

setUTCSeconds()

Exactly like **setSeconds()**, but based on UTC.

```
d.setUTCSeconds(30)
console.log(d) // 2020-07-22T21:46:30.983Z

d.setUTCSeconds(30, 452)
console.log(d) // 2020-07-22T21:46:30.452Z
```

setUTCMilliseconds()

Exactly like **setMilliseconds()**, but based on UTC.

```
d.setUTCMilliseconds(-230)
console.log(d) // 2020-07-22T21:47:34.770Z
```

Conversions

There are many ways to retrieve a date after setting their values or leaving the current date to be:

toString():

```
console.log(d.toString()) // Thu Jul 23 2020 02:37:39 GMT+0430 (Iran Daylight Tim
e)
```

toDateString():

```
console.log(d.toDateString()) // Thu Jul 23 2020
```

toTimeString():

```
console.log(d.toTimeString()) // 02:38:15 GMT+0430 (Iran Daylight Time)
```

toUTCString():

```
console.log(d.toUTCString()) // Wed, 22 Jul 2020 22:08:37 GMT
```

toISOString()

```
console.log(d.toISOString()) // 2020-07-22T22:03:14.131Z
```

toJSON():

```
console.log(d.toJSON()) // 2020-07-22T22:04:14.477Z
```

toLocaleString():

```
console.log(d.toLocaleString()) // 7/23/2020, 2:37:10 AM
```

toLocaleDateString():

```
console.log(d.toLocaleDateString()) // 7/23/2020
```

toLocaleTimeString():

```
console.log(d.toLocaleTimeString()) // 2:36:26 AM
```

Chapter 13: NPM

As mentioned before, programming sometimes requires you to use the codes written by others. Reinventing the wheel is something that no one likes, especially programmers! As you've read earlier about code splitting and exporting or importing modules, it is possible to use the functionality of a different file inside another file, which gives you the power to organize your codes. Node.js supports a way to handle packages that get installed using a special tool called **npm** that stands for **Node Package Manager**. By using **npm**, you can define your project and also its dependencies. In this chapter, we'll look into some most useful commands this tool provides us with.

To start a Node.js project, you need to run the command below in the root of your project:

```
$ npm init
```

This command walks you through some steps in order to get your project read and creates a file called **package.json,** which holds the information you provided. This file also registers the dependencies needed for your project. A **package.json** file looks like this (not exactly):

```
{
  "type": "module",
  "name": "nodejs-essentials",
  "version": "1.0.0",
  "description": "A good way to start learning Nodejs",
  "main": "app.js",
  "author": "Adnan Babakan",
  "license": "MIT"
}
```

As the file extension indicates, this is a JSON formatted file.

npm install

npm install is the command you are going to use the most. This command installs a specific package from the **npm repository**. You can search for **npm** packages using Google or by going to https://npmjs.com.

Here is how the command looks like:

```
$ npm install {package_name}
```

There is a package called **random** available at https://www.npmjs.com/package/random, which helps us to create random numbers. To install this package, you simply need to run the command below:

```
$ npm install random
```

There is also a short way to install packages:

```
$ npm i random
```

After the process is finished, you'll see a new folder called **node_modules** in your project's root directory. This directory is where your packages are. Now that you have installed this package, you can use it by importing it. Importing a package requires its name only and not the path to its file, like when it is needed for self-written files.

```
const random = require('random')
```

If you run the command below, along with installing the package, it gets added to your **package.json**:

```
$ npm i random --save
```

Tip: Anything prefixed with a dash (-) or double dash (--) is called a flag.

After running this command, your **package.json** should look like this:

```
{
  "type": "module",
  "name": "nodejs-essentials",
  "version": "1.0.0",
  "description": "A good way to start learning Nodejs",
  "main": "app.js",
  "author": "Adnan Babakan",
  "license": "MIT",
  "dependencies": {
    "random": "^2.2.0"
  }
}
```

If your package is not used in production is only meant for development you can install it as a dev dependency using the **--save-dev** flag:

```
$ npm i random --save-dev
```

Which makes this package a dev dependency and registers it as so:

```
{
  "type": "module",
  "name": "nodejs-essentials",
  "version": "1.0.0",
  "description": "A good way to start learning Nodejs",
  "main": "app.js",
  "author": "Adnan Babakan",
  "license": "MIT",
  "devDependencies": {
    "random": "^2.2.0"
  }
}
```

If you are asking what's the reason to submit dependencies in **package.json,** here is the answer: The **node_modules** folder can be hefty, and you should not ship it with your application (neither on the server nor to another computer). Instead, you'll leave your **node_modules** folder, and when your application is in a new environment, what you need to do is running this command:

```
$ npm i
```

What this command does is pretty simple! It installs all the dependencies mentioned in the **package.json**. If you only want to install the production dependencies (exclude the dev dependencies), you should use the **--production** flag:

```
$ npm i --production
```

If you only want to install dev dependencies, you can do so by the command below:

```
$ npm i --only=dev
```

Npm is not limited to its repository and can install packages from GitHub or even from a tarball file (.tgz):

GitHub samples:

```
$ npm i user/repo
```

```
$ npm i user/repo#branch
```

You can install from GitLab as well:

```
$ npm i gitlab:user/repo#branch
```

Tarball file samples:

```
$ npm i ./tarballfile.tgz
```

```
$ npm i https://example.com/tarballfile.tgz
```

Versioning

Npm uses a specif way to version packages, and you can use it to install a specific version of the package you need by indicating the version after the **@** symbol.

This command installs the latest version:

```
$ npm i {package_name}@latest
```

A number can be specified as well:

```
$ npm i {package_name}@2.0.0
```

A version range is also supported:

```
$ npm i {package_name}@"2.3.0 – 2.3.4"
```

A more advanced range indicator is also possible to use:

```
$ npm i {package_name}@">=2 <3"
```

By using the carat sign (^), you can tell npm you want versions greater than the specified version in the same major range. For instance, ^2.2 shows that you want any version from 2.2 to 3.0 (not included).

```
$ npm i {package_name}@^2.2
```

The tilde sign (~) gives similar functionality as the carat sign, but it includes the range in the same minor range. For instance, ~2.2 indicates that you want any version from 2.2 to 2.2.x.

```
$ npm i {package_name}@~2.2
```

Versions can be combined with double pipe (||) sign to indicate multiple sets of versions:

```
$ npm i {package_name}@"~2.2 || ~4"
```

Global packages

Global packages are the ones that got installed in the **node_modules** folder where your **npm** tool's program is installed. These packages are available through the entirety of your system. A global package can be installed using the **-g** flag:

```
$ npm i {package_name} -g
```

Caution: Only packages that provide tools or executables should be installed globally. Like CLI tools. Project-specific packages should be installed locally, so they match the version they need and avoid conflicts.

npm remove

This command acts exactly the opposite of **npm install**. By using this command, you can remove a package:

```
$ npm remove {package_name}
```

For instance, if you want to remove the **random** package, you can run this command:

```
$ npm remove random
```

This will delete the package from the **node_modules** folder, but it will still be present as a dependency in the **package.json** file. If you want to remove it from there as well, you should use the **–save** flag:

```
$ npm remove random --save
```

And if you've installed this package as a dev dependency, you should use the **--save-dev** flag to remove it from the **package.json** file:

```
$ npm remove random --save-dev
```

npm list

This command lists the packages of the current project.

```
$ npm list

`-- random@2.2.0

  +-- babel-runtime@6.26.0

  | +-- core-js@2.6.11

  | `-- regenerator-runtime@0.11.1

  +-- ow@0.4.0

  +-- ow-lite@0.0.2

  `-- seedrandom@3.0.5
```

By adding the **-g** flag, you can list the globally installed packages:

```
$ npm list -g
```

Chapter 14: File System

One of the essentials of every program is working with files. It can be used to read information to a file or to write them into a file. As you've read before in previous chapters, the file system of Node.js is functioning thanks to the **fs** module. It is needed to import the file system before going any further:

```
const fs = require('fs')
```

Now it is possible to access all the methods used to work with files using the **fs** constant. This module features two kinds of methods, which are **asynchronous** and **synchronous**. The asynchronous methods take a callback to be called when the operation is done. On the other hand, synchronous methods (which are suffixed with the **Synch** keyword) will block the thread and won't let Node.js proceed until the operation is finished. Using synchronous methods is highly discouraged.

File

Read

Reading is a common task done using the file system. This goal can be achieved by using the **readFile** or **readFileSynch** methods. The signature of **readFile** is as below:

fs.readFile(path[, options], callback)

The first argument is for specifying the path of the file that is to be read, and the optional second argument is to set the encoding or the flag (file system flags). The callback is to get the data or the encountered error.

```
fs.readFile('test.txt', 'utf-8', (err, data) => {
    if(err) console.log(err)
    else console.log(data)
})
```

By running the code above, you'll see the data of the file in the command line or an error as below if the file doesn't exist:

```
[Error: ENOENT: no such file or directory, open 'C:\Book samples\test.txt'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\\Book samples\\test.txt'
}
```

If you skip the second argument and try to log the data, you'll have a result like below:

```
<Buffer 48 69 20 74 68 65 72 65 21>
```

This is a buffer data, and encoding (charset) is needed to convert this into a readable form. The second method used for reading files is **readFileSynch,** which is a synchronous, blocking method. The arguments are the same as **readFile** except for the callback, which this method doesn't need one.

```
let data = fs.readFileSync('test.txt', 'utf-8')
console.log(data)
```

Since this method doesn't have a callback, if there is an error reading the file specified, it will throw an error, and you need to use a try-catch block to handle it.

```
let data

try {
    data = fs.readFileSync('test.txt', 'utf-8')
} catch(e) {
    console.log(e)
}

console.log(data)
```

Write

Using a write method will allow you to alter the data inside a file. There two methods called **writeFile** and **appendFile** used for this purpose, which both have their synchronous counterparts as well. Here is the signature for both methods:

fs.writeFile(file, data[, options], callback)

fs.appendFile(file, data[, options], callback)

The **writeFile** method will allow you to erase the data inside a file and write new data inside it. The callback in these functions takes an argument showing the error if there are any.

```
fs.writeFile('test.txt', 'Hello world', err => {
    if(err) throw err
    console.log('Done!')
})
```

If the **options** argument is a string, it represents the encoding:

```
fs.writeFile('test.txt', 'Hello world', 'utf-8', err => {
    if(err) throw err
    console.log('Done!')
})
```

If you don't want to wipe the data inside the file and append your data to it, you can use the **appendFile** method, just like the **writeFile** method:

```
fs.appendFile('test.txt', 'Hello world', 'utf-8', err => {
    if(err) throw err
    console.log('Done!')
})
```

If the file doesn't already exist, these methods will create such a file.

Rename

To rename a file, we can use the **rename** method, which its signature is as below:

fs.rename(oldPath, newPath, callback)

The **oldPath** argument is the current path to the file, and the **newPath** is where it should be. Renaming a file is like moving a file.

```
fs.rename('a.txt', 'b.txt', err => {
    if(err) throw err
    console.log('Done')
})
```

If a directory is available, you can move the file into that directory:

```
fs.rename('b.txt', 'my-files/b.txt', err => {
    if(err) throw err
    console.log('Done')
})
```

Delete

Deleting a file is easy with **fs**. You only have to use the **unlink** method. The signature is as below:

fs.unlink(path, callback)

```
fs.unlink('b.txt', err => {
    if(err) throw err
    console.log('Done')
})
```

Directory (folder)

Node.js file system allows you to work with directories as well as files.

Open

There are two methods for this particular task, which are **opendir** and **opendirSync,** and their signatures are as below:

fs.opendir(path[, options], callback)

fs.opendirSync(path[, options])

```
fs.opendir('my-folder', (err, dir) => {
    if(err) throw err
    console.log(dir.constructor.name) // Dir
})
```

As you can test by yourself, the callback has an argument called **dir,** which is of type **Dir**. The synchronous method returns such an object as well:

```
let dir = fs.opendirSync('my-folder')
console.log(dir.constructor.name) // Dir
```

Since this is a blocking method, we won’t be using it.

The **options** argument can be a string or an object representing the encoding

You can get the full path for your directory using the **path** property:

```
fs.opendir('my-folder', (err, dir) => {
    if(err) throw err
    console.log(dir.path) // my-folder
})
```

This might seem redundant, but it is useful when you don’t know which directory is open such as when it is asked from the user to specify the folder.

Read

You can read a directory to return its content. You can use the **readdir** or its synchronous counterpart:

fs.readdir(path[, options], callback)

The callback gets two arguments, one for the error as the convention, and the second one holding the content of the directory in an array:

```
fs.readdir('my-folder', (err, content) => {
    console.log(content) // [ 'another-folder', 'b.txt' ]
})
```

Create

To create a directory, you can use the **mkdir** method and its synchronous counterpart.

fs.mkdir(path[, options], callback)

```
fs.mkdir('new-folder', err => {
    if(err) throw err
    console.log('Done')
})
```

Rename

Renaming a folder is just as renaming a file.

```
fs.rename('folder-one', 'folder-two', err => {
    if(err) throw err
    console.log(err)
})
```

Delete

Deleting a directory is a simple task. You can use **rmdir** method and its synchronous counterpart to delete a directory when it has no content in it.

fs.rmdir(path[, options], callback)

```
fs.rmdir('new-folder', err => {
    if(err) throw err
    console.log('Done')
})
```

If you try to delete a non-empty directory you will encounter an error:

```
fs.rmdir('my-folder', err => {
    if(err) throw err
    console.log('Done')
})
```

The code above will generate this error:

```
[Error: ENOTEMPTY: directory not empty, rmdir 'C:\Book samples\my-folder'] {

  errno: -4051,

  code: 'ENOTEMPTY',

  syscall: 'rmdir',

  path: 'C:\\Book samples\\my-folder'

}
```

If the directory is not empty, you should set an option called **recursive** true**,** so the file system deletes everything inside the directory as well as the directory itself.

```
fs.rmdir('my-folder', {recursive: true}, err => {
    if(err) throw err
    console.log('Done')
})
```

Resources

Books

Algorithms by Robert Sedgewick and Kevin Wayne

Websites

https://developer.mozilla.org

https://www.w3schools.com/js/

https://www.tutorialsteacher.com/

https://eloquentjavascript.net/

https://www.techotopia.com/

https://nodejs.org/