**Introduction to Programming with JavaScript**

So much of what you take for

granted in your everyday online life,

like using a search engine, shopping,

and using social media,

is made possible by JavaScript.

Nearly all of the interactivity you experience online,

like drop-down menus or the autocomplete function,

is facilitated by JavaScript.

JavaScript is most well-known for

being the programming language of the web.

But it has become so much more.

Now, it is used not only for the front-end development,

but also for back-end development,

and you can even create

mobile apps using JavaScript and React.

Without JavaScript,

the modern software development landscape

would look a lot different.

Because it is such a powerful tool

and so integral to software development,

it is important for you,

as a new developer,

to know how it works and how to use it.

In this course, starting with Module 1,

you will cover an introduction to JavaScript,

that will help you understand how

it is used in the real-world.

You'll learn how to set up

Visual Studio Code and

install the code runner extinction.

This module also introduces

you to programming in JavaScript,

which includes learning about variables, datatypes,

operators, strings, booleans, and numbers.

After this, you'll progress to learning

about conditional statements and loops.

In Module 2,

you will build on what you've

learned in the first module,

by learning about arrays, objects, and functions.

You will also learn about bugs and errors.

In Module 3,

you will learn about the functional programming

and object-oriented programming paradigm,

which includes function calling, recursion, and scope.

With scoping, you will learn how to

use var, let, and const.

Furthermore, you will learn

about object-oriented programming,

and you will design and build an object-oriented program.

You will also learn about advanced JavaScript features,

like destructuring arrays and objects

using four of loops and template literals.

Additionally, you will learn about

data structures and the spread and rest operators.

You will also learn how

JavaScript is used in the browser.

In this module, you will have an opportunity

to practice updating the contents of a webpage.

Module 4 is all about testing and compatibility.

Therefore, you will learn about types of

testing and you'll write a unit test.

You will also be introduced to Jest and

learn how to write tests with Jest in an exercise.

You will learn about test-driven development

and get to practice applying it too.

You will also learn about JavaScript testing challenges,

which include learning about webpack,

transpiling, and working with arrow functions in React.

As part of this module,

you will also learn about other JavaScript environments,

like Node and NPM.

In Module 5,

you will have an opportunity to put your new skills into

practice by creating the little lemon receipt maker.

In summary, this course provides

you with an introduction to JavaScript.

It is part of a program of courses that

leads you towards a career in software development.

You have probably encountered

many new technical words and terminology in this video.

Don't worry if you don't fully

understand all these terms right now,

everything will become clear soon.

After all, this is part of

the reason why you were taking this course.

**Why JavaScript?**

In the last few hours or maybe even in the last few minutes,

you've probably interacted with web pages to do things like watch videos, look for

locations on maps, or interact with your social media feeds.

In fact you're doing it right now with this course.

But what makes this interactivity possible?

The programming language JavaScript.

Let's find out a bit more about the benefits of JavaScript and

its importance to developers.

JavaScript is a language that builds interactivity into web pages.

It is literally the language of the web.

Almost every website runs JavaScript in some form or another.

The reason for this is the fact that since its very inception In 1995,

JavaScript has been the main way to interact with web pages on

the client side, the front and side of websites and web applications.

Using JavaScript updates are displayed in real time on our devices.

Some common examples include interactive maps and client side form validation.

Over the years, there have been some alternatives to JavaScript,

such as VBScript and more recently TypeScript.

But even typescript compiles down to JavaScript so browsers can understand it.

Since JavaScript is one of the most widely available and frequently used methods for

interacting with the browser on the client side,

the language became immensely popular.

In fact, developer surveys often show that JavaScript remains one of the most popular

programming languages today.

So what is it that makes JavaScript so integral to modern browsers.

JavaScript is currently the only computer language that

allows us to directly interact with our web pages dynamically on the client.

It's baked into the browser.

Its enduring popularity is down to the rules of backwards compatibility, which

states that all websites that were built in the past still need to work today.

So removing JavaScript from the browser would effectively render millions of

websites completely useless.

This means that JavaScript is in the unique position of being one of several

central pillars of web development.

You've just discovered the importance of JavaScript to web development.

JavaScript unique ecosystem is one of the main reasons for its popularity.

Simply put, browsers speak JavaScript.

But there are a few other important reasons why we should embrace JavaScript.

One of the main reasons that developers use

JavaScript is because of how easy it is to use.

There are many programming languages where a newcomer needs to do a substantial

amount of prep work to even get set up and ready to learn.

However, with JavaScript it's as simple as opening the browser's developer tools and

navigating to the console tab because every browser has a JavaScript engine

built in and you can interact with it using the console.

There are still a few small steps required to get started with JavaScript.

These include installing a code editor and

possibly setting up some other things such as Node.js and NPM.

But initially as a beginner,

it's enough to just open the developer tools on your web browser.

Another reason to learn and use JavaScript is because as we saw earlier,

it's everywhere.

JavaScript is used in almost every website there is.

It's used on the client side of sites as plain JavaScript also known

as vanilla JavaScript.

It fuels a myriad of JavaScript frameworks such as React, Vue,

and D3 and it's also used on the server as Node.js and more recently Dino.

Another reason why people often choose JavaScript is that it's considered one

of the more accessible programming languages.

It also has a wide development community to offer help and guidance.

This is especially true when it's compared to other languages.

There are many languages that for

a variety of reasons would be very challenging to begin programming with.

But because it's so approachable, JavaScript can be a great way for

new developers to jump into programming.

So besides being a useful skill on its own,

JavaScript can also be a stepping stone to other languages and technologies.

Finally, one more reason to learn JavaScript is because JavaScript skills

are in demand.

Countless job postings appear for JavaScript developers every day and

these will keep coming for the foreseeable future.

So next time you ask why choose JavaScript remember that it's the foundation

of technology of everything you do online with a huge variety of uses.

Plus it's easy to get started with and once you're up and

running there's lots of job offers and projects open to you.

Best of luck with your JavaScript journey.

**Programming in JavaScript**

Javascript is used in many scenarios, for example,

in the browser to help add

various behaviors and interactivity,

like adding an item to

a shopping cart when you click a button.

On the server, it can be used to power up websites,

communicate with databases, and

provide a native fields to web apps.

It's used to build mobile apps using technologies like

React Native and it's used to

program devices known as the Internet of Things.

In a nutshell, JavaScript is everywhere.

This means that there are many different use cases

for JavaScript and with these different use cases,

come different implementations,

each geared towards solving a particular problem.

For example, in the early 2000s,

different companies built Internet browsers

and were referred to as browser vendors.

However, with different browsers came

different behaviors with

various discrepancies between other browsers.

This resulted in developers sometimes having to write

separate JavaScript code for different browsers.

This wasn't the best use of developer time and

could lead to a frustrating experience for end-users.

Out of this frustration,

several projects appeared trying

to solve this compatibility problem.

One such project managed to solve

these issues and it was a library named jQuery.

Using this library, all a developer needed was to

import jQuery and write code using its features.

This code would then work across all browsers.

At the time, it was like magic,

as it was such a great solution to

the problem of incompatibility and

jQuery became the most popular JavaScript library

for well over a decade.

However, as the web and the way we code kept evolving,

new problems appeared, and soon enough,

new solutions to those problems needed to be built.

Once such solution was React,

which came out in 2011 and it had

a considerable effect on the way we

think about websites and how we build them.

React, solved many of

the issues associated with creating,

updating and maintaining complex websites.

Soon, many other technologies

appeared attempting to do the same.

These include Knockout, Backbone,

Angular, Ember, Vue,

Alpine, and others.

With millions of websites containing

JavaScript code from different versions and libraries,

there is a lot of old code.

This is known as a legacy code.

While you probably won't use

jQuery to build a modern website today,

you might still come across it in

a project that is still actively running.

But don't worry, sometimes

beginners think they have to learn or

even master all the different technologies

associated with JavaScript.

However, that's not really necessary.

To be a well-rounded developer,

you need to learn and master the basics of

plain JavaScript without the frameworks.

Once you have this foundation,

the pathway will become

easier for you to learn a framework

built on top of JavaScript, such as React.

**Writing your first Javascript code**

In this reading, you'll learn about comments in JavaScript. Additionally, you'll learn about the semi-colon in JavaScript: what it does and why it is used. You will then download a browser if you don't have one installed and run your first piece of JavaScript using the Console.

**Comments in JavaScript**

I've chosen comments as the starting point for two reasons:

1. Their syntax - the way comments are written is easy to understand.
2. Writing comments can empower you as a developer.

First, I'll explain the syntax, and after that, I'll discuss why being able to write comments is so empowering.

**Comments in JavaScript: the syntax**

There are two varieties of comments in JavaScript:

1. Single-line comments
2. Multi-line comments

A single-line comment is created when you add two forward-slash characters one after the other, without spaces.

// this is a comment!

Anything that follows a single-line comment in JavaScript is ignored by the browser.

This means that, essentially, you can write any kind of text, code, characters, emojis, whatever - and the browser will ignore it.

A multi-line comment, as its name says, spans for several lines of code and is created with a forward slash and a star. For example:

/\*

this

is

a

multi-line

comment

\*/

You can also use the multi-line comment syntax on a single line of code, as follows:

/\* this is a multi-line comment on a single line \*/

**Why writing comments is empowering**

In this course, it is assumed that you've never written a single line of JavaScript code.

With this assumption in mind, consider the effects of what you've just learned, that is, the effects of learning how to write comments in JavaScript:

1. You can now freely express your ideas about any code that you write.
2. You can add comments to any code that already exists.
3. Those comments can be intended for your future self, or for colleagues on your development team.

So, comments are empowering because they facilitate communication with your future self or with your team members, allowing you to ask questions about the code, mark the code as "to do", or as "to improve", or just simply explain what a given piece of code does.

Additionally, you can even comment out some working code in a JavaScript file - to prevent it from running.

Effectively, comments allow you to "switch off" pieces of JavaScript code.

There can be many reasons for that:

1. Trying to understand how a given piece of code works.
2. Testing different solutions to a coding problem - while not having to delete existing code.
3. Debugging - trying to pin-point why your code is broken or behaving unprediticably.

**The semi-colon in JavaScript**

In the English language, the fullstop or period - the *.* character - is used to separate thoughts into sentences.

By clearly separating thoughts with the fullstop, you avoid being misunderstood.

In JavaScript, the semi-colon - the *;* character - has a similar purpose: it is used to clearly delimit parts of the code from some other parts of the code.

**Automatic Semi-Colon Insertion (ASI)**

Interestingly, the browser has a feature known as "Automatic Semi-colon Insertion" - meaning, it does a pretty good job of "filling in the blanks" in case there is a missing semi-colon where there should be one.

Effectively, what that means for developers is that most of the time, it makes no difference if a semi-colon is added or not, since the browser is likely to figure it out anyway.

That's why some developers say that you shouldn't bother with adding semi-colons at all.

However, other developers argue that it's better to use it wherever it's needed - for the sake of clarity.

The truth is that most of the time, you can think of adding semi-colons in JavaScript as optional - and somewhat of a stylistic preference.

**A note on using the console in the developer tools in your browser**

As already mentioned earlier on in this course, one of the reasons why JavaScript is so popular is because it's so approachable.

To get started with JavaScript, all you need is a browser. In this course I'll be using Google Chrome.

Once you've installed the browser and run it, right-click on the currently active web page and click the *Inspect* command on the right-click contextual menu.

This will open the Developer Tools and then you can click on the Console tab to open the console, or alternatively, pressing the *ESC* key will toggle on and off the console regardless of the currently active Developer Tools panel.

You can type any JavaScript command you like into the DevTools console.

**If you need to type multiple lines of code before you run them, make sure to press the SHIFT + ENTER shortcut key to get onto the next line.**

Notice the distinction between pressing the *ENTER* key to run the JavaScript code you've typed, versus pressing the *SHIFT + ENTER* shortcut to move onto the next line of code (rather than running the code you've already typed up).

This is all that you need to get started writing JavaScript code!

In the upcoming lessons, feel free to follow along in either of two ways:

1. Using the VS Code editor and the Code Runner extension as previously described
2. Using the Chrome browser itself, and running the code inside the DevTools console as described in this reading

**Output a greeting into the console**

Now that you know how to get to the Developer Tools' Console tab, let's now use it to run your first piece of real JavaScript code.

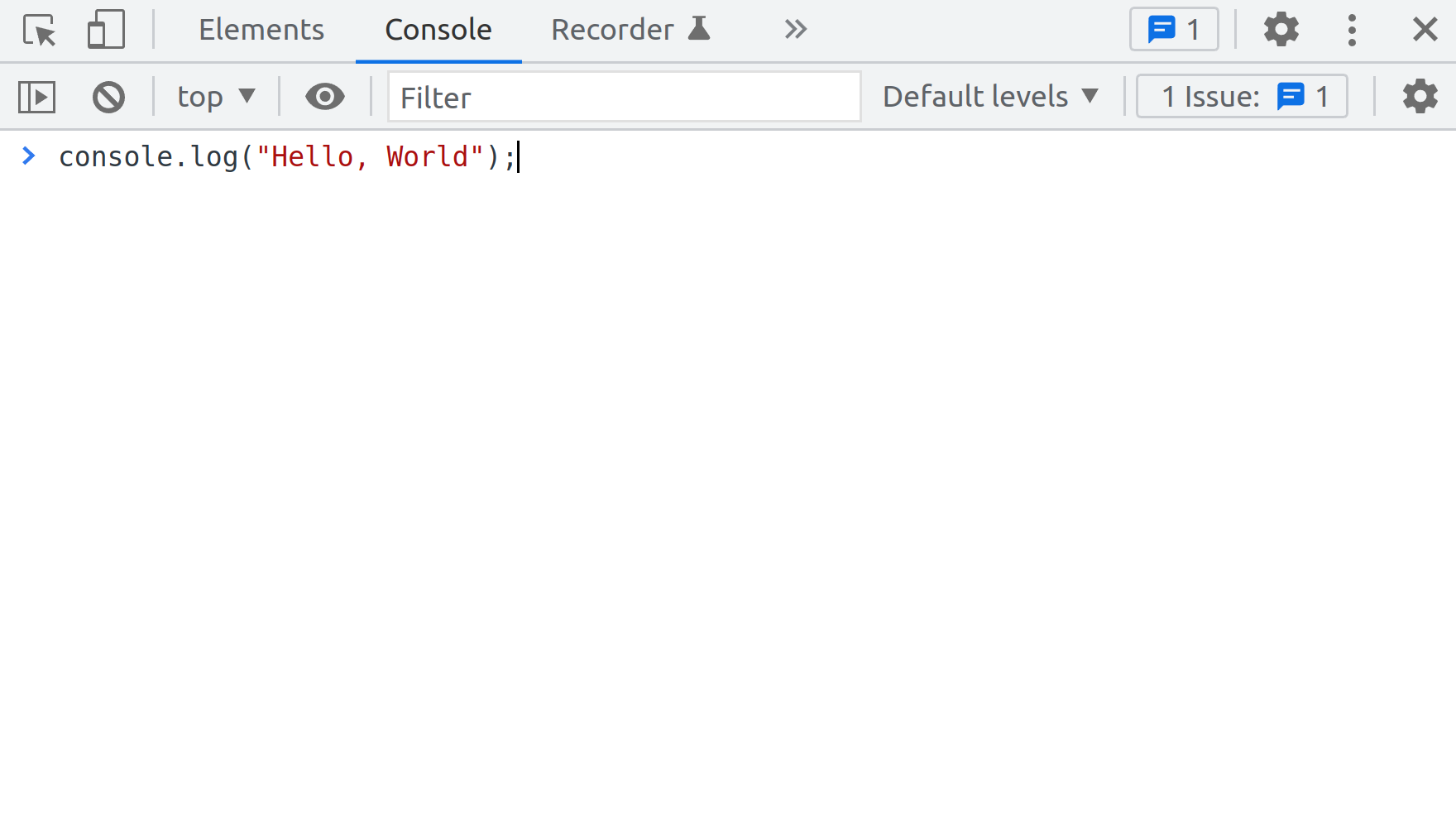
In Chrome, with the Developer Tools open, click into the empty space in the console tab, just to the right of the blue *>* character. You should see a blinking vertical line (also referred to as "the cursor"). The cursor indicates that you can type into the console.

If you type valid JavaScript code, it will be executed, meaning: it will be processed and it might result in some kind of output.

You'll now use the *console.log* function to output the words "Hello, World".

To do so, type the following command into the console:

console.log("Hello, World");



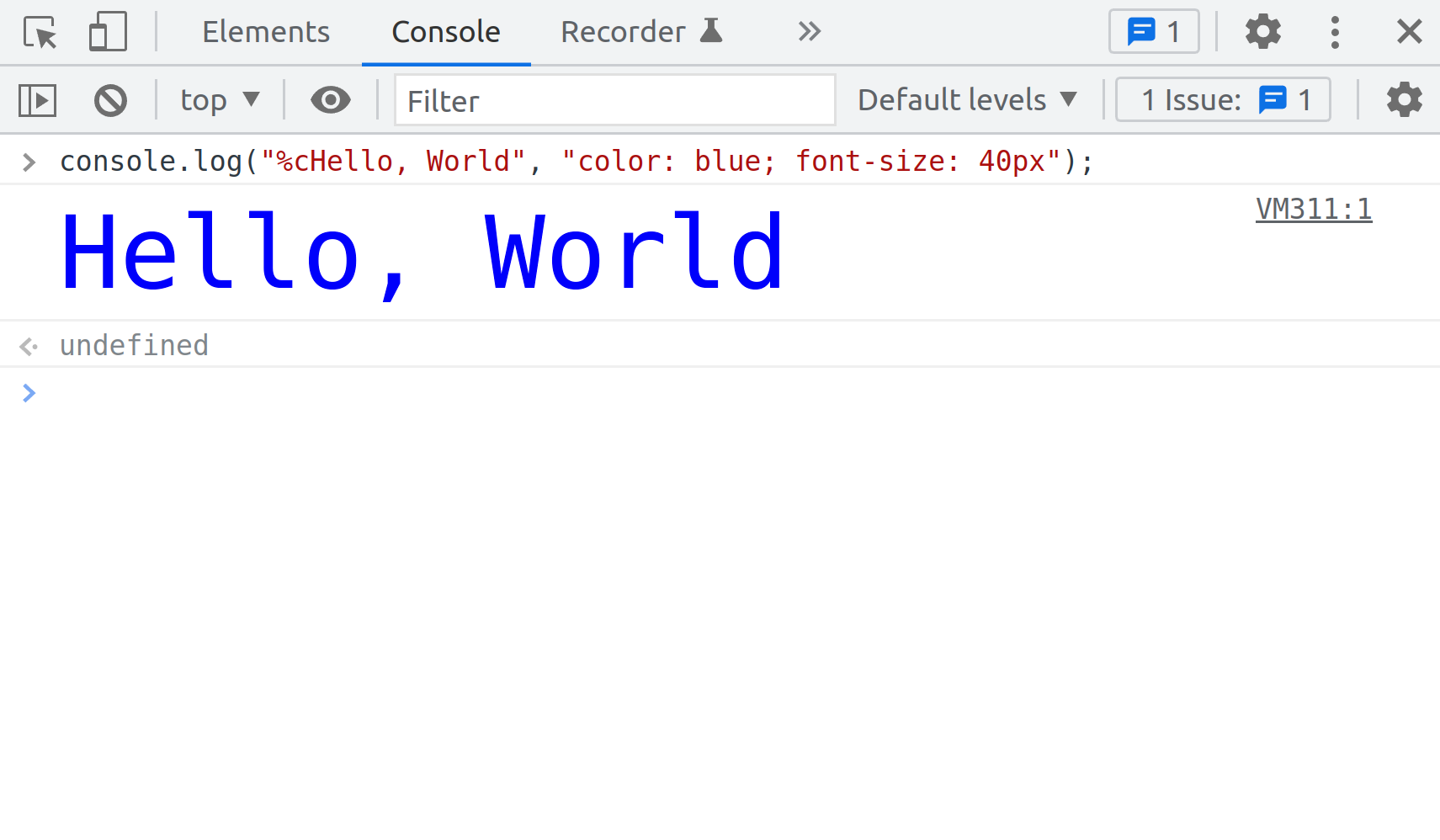
If you've done everything as instructed, the words "Hello, World" should be output in the console.

Here's another, more complex command, to show you that the *console.log* command comes with a number of tricks.

For example, did you know that you can style the output in the console?

In this code snippet, there are a few additions: the font size is different and the color is blue:

console.log("%cHello, World", "color: blue; font-size: 40px");



If you copy-paste this piece of code, or perhaps, simply type it into the console, once you press the *ENTER* key to run it, you'll get the words "Hello, World" output to the console. This time, however, the color of the letters will be blue, and the font size will be 40px. So, you've just learned a nice trick with the console.

If you add the *%c* right after the *"* character, you can then style the console output by adding the *,* character after the second *"*, and then, inside another pair of *"* and *"* characters, use valid CSS code to style the words you want to output in the console.

The reason for showing you this little trick was to hopefully get you motivated to practice writing various words into the *console.log* command, and to use your CSS skills to change the styling of these words in the console output. That way, you might find it fun to practice this newly acquired skill - and learning and fun always go nicely together.

**Output multiple words into the console**

To output multiple words into the console, you can join them using the *+* character, formally known as the "concatenation operator" when we're joining pieces of text, or the "addition operator", for performing the mathematical operation of adding two numbers.

Here is an example of joining three separate pieces of text: *console.log("Hello " + "there, " + "World")*. The output of this command will be: *Hello there, World*.

H​ere is an example of outputting three separate pieces of text using the , character instead:

*console.log("Hello ", "there, ", "World")*

T​he output of this command will still be: *Hello there, World*.

**Data types**

Imagine for a moment that you are faced with the task of moving all your

belongings to a new place.

What steps would you take to make the process more efficient?

Well, you probably start by sorting items and packing them into boxes and

then labeling those boxes so that you have an idea of what each one contains.

After moving them,

you put each box in the room it belongs to make unpacking quicker and easier.

You can also think of programming in a similar way to packing and moving boxes.

You have many different data types at your disposal but

coding efficiently starts with knowing when and where to use them.

In this video you learn to identify the primitive data types in JavaScript.

You will learn about their specific use cases and when to use each data type.

There are seven primitive data types in JavaScript.

These are string, number, Boolean, null, undefined, BigInt and symbol.

Let's begin by learning about the first two which are strings and numbers.

Let's say that you are building an E-commerce app for selling guitars.

When a user views the page for the most popular model,

you want them to see the name of the guitar, the description,

the best guitar around and the price of 375.

Seems simple enough, right?

But how can you ensure that the information displayed on screen is stored

effectively so that the app can use it?

You would do this by using the most appropriate data types but

what does data mean exactly?

Well, each piece of information in your app has a different value and

all values are collectively referred to as data but the values differ and

therefore need to be stored differently.

For example, you store the name and description as text values and

the price as a numerical value.

In JavaScript, text values are known as the string data type while

numerical values are referred to as the number data type.

From a programming perspective, making a distinction between numbers and

text is important because they adhere to different rules.

For example, to build a number, you simply type in the numerical values but

to build a string in JavaScript,

the characters must be enclosed in either single or double quotes.

There is also a difference in the number of combinations that the number and

string data types can store.

The number data type has a very wide range in JS enough for most common use cases.

However, it is limited up to a point determined by JavaScript calculation

capabilities.

The string data type practically has an unlimited number of combinations of

characters.

There is almost an infinite number of ways we can combine different characters

into strings.

Okay, so back to our app example, the data must be stored using different types.

The price will be stored as a number while the name and

description will be stored as strings with each one wrapped in double quotes.

You now know that the string data type is great for storing values such as titles

and descriptions while the number data type is better suited for

prices or any value that you would like to calculate.

However, there are additional data types in JavaScript to help you tackle

different tasks.

Next let's explore each of the remaining data types and their specific use cases to

better understand how you might make good use of them.

Let's think of each one as a box.

Like I mentioned earlier, the Boolean data type has only two values, true and false.

This means that it is useful for making decisions.

Sometimes it's necessary to know when a variable does not contain a value.

And JavaScript has two data types to express just that.

First, there is the null data type which only has the value null and

represents the absence of value.

You also have the undefined data type which can only hold the value undefined

and usually refers to a variable that has not yet been assigned a value.

The capabilities of JavaScript have evolved over time and version ES6

introduced two new primitive data types to help with more complex tasks.

One is the BigInt data type which is like an extra large box that can

accommodate a much greater range of numbers than the number data type.

Finally, there is the symbol data type which can be used as a unique identifier.

Think of it as having multiple boxes with the same label and

using different serial numbers to avoid mixing them up.

In this video,

you've learned about the various data types available in JavaScript.

Remember that every data type has a specific use case and

knowing when to apply each one will make you a better coder, see you next time.

**Operators**

Where do you think the name operators come from?

Does it have something to do with

an operation of some sort?

Well, the answer is yes.

Operators are used to perform

operations on variables and values.

An operator is used to manipulate

individual data items and return a result.

For example, you may already be familiar with

the assignment operator to assign a value to a variable.

You use assignment operators for

simple and complex calculations.

They provide us with

similar options to those on a calculator.

In this video, you will learn about assignment and

logical operators and how to use them in JavaScript.

First, let's start with

the assignment operators which you can

use to perform simple arithmetic.

For example, to add two numbers,

you can use the addition operator,

which is represented by the plus symbol.

If you want to subtract values,

use the minus operator,

which is represented by the hyphen symbol.

If you want to perform division, for example,

35 divided by 5,

use the division operator,

which is represented by the forward slash.

Finally, to multiply numbers,

use the multiplication operator,

which is represented by the star symbol.

In addition to assignment operators,

you can also use comparison operators,

which compares and returns

a logical value based on whether the comparison is true.

For example, you can compare

numbers to check if they are greater than,

less than, equal or not equal to each other.

You will learn more about the various assignment and

comparison operators later in this course.

But for now, let me tell you about logical operators.

Logical operators are used in JavaScript

to determine if something is true or false.

For example, the logical operator named end,

which you can use to check

the two or more conditions are true.

Imagine you had a variable named A and you

wanted to check that its value is

greater than five and less than 10.

For this to be satisfied,

both statements must be true.

Alternatively, there is the operator named R,

which checks for at least

one of the statements to be true.

For example, A is greater than five,

RA is greater than 10.

Finally, you can use the logical operator,

NOT, which returns a false value if the result is true.

Operators are used to control

the flow of a program that meets certain criteria.

Now let me demonstrate how to use JavaScript,

math and logical operators.

I've opened VS Code here.

Now, let me demonstrate how you can use

arithmetic and comparison operators in JavaScript.

Recall that comments are used to

specify which lines JavaScript should ignore.

As you learned earlier,

you can put almost anything inside comments.

Notice I have two multi-line comments.

The first one lists some arithmetic operators,

and the second one lists some comparison operators.

I also have several single-line comments.

You already learned that

JavaScript completely ignores comments,

and here's the proof.

I click on the Run code icon.

Notice that the output in the right panel will not show

any values other than

the expected additional info on the code that was run.

Let me clear that for now and show you how to work

with some arithmetic and comparison operators.

I'm going to uncomment

all single-line comments from lines 14-21,

so that you can focus on each of

these console logs separately.

This will then allow me to demonstrate

how each of these operators behaves.

Let me delete the two forward slashes on

line 14 and rerun this file.

Notice I get the value of four outputted to the console,

resulting from adding the values 2 plus 2.

Let me clear the console,

comment line 14 and uncomment line 15.

This is an example of adding

multiple numbers using the plus operator.

Notice when I run this,

I get the output of 15.

Let's do another one using

the subtraction operator to subtract 18 from 20.

If I run this, I get the expected output of two.

Next is multiplication,

which is represented by the asterisk symbol.

Here I will console.log the values

of 2 times 3 to get the value of six.

On line 18, I use

the forward slash symbol

for the division of eight by one.

Notice when I run this,

a value of eight is outputted, success.

That's it for performing simple arithmetic in JavaScript.

Let me clear my screen again.

Now I want to demonstrate how to

work with comparison operators.

Notice on line 19,

I use the greater than symbol to

check if three is greater than two,

run it and indeed it is,

I get an output of true.

On line 20 I use the greater than

symbol to check if two is greater than three.

Since it's not, I receive an output of false.

These words true and false are

special kinds of values used in JavaScript.

Don't worry too much about them now,

as you will learn more about this later.

For now, I just want you to

focus on the final line of code.

This checks if one value is equal to another value.

It's important to remember that you need to use

the equals symbol twice for

this comparison operator to be valid.

This is because the single equal symbol

is used for the assignment operator,

like when you assign a value to a variable.

I would expect the value of 10 to be equal to 10.

Let's run it to find out and it is.

Notice a value of true is outputted to the console.

Congratulations, you have now

learned how to work with operators in JavaScript.

This is a must-have skill in your development toolkit

as they are used in many areas

of programming. Great work.

**Numbers**

In JavaScript, like in other programming languages,

there are different data types.

The number of data type is

a foundational part of JavaScript as

a programming language because it represents

both integer and decimal point numbers.

Without it, you wouldn't be able to code and with

such a crucial data type

we better get started right away.

Let me demonstrate the number data type.

I type the value 123

into the console and press ''Enter.''

Notice that the output is 123.

Next, on a new line,

I type a decimal value,

123.456 which is also a number data type.

I press the Enter key again and get

the decimal value 123.456.

In JavaScript, we can use

numbers to perform math operations.

For example, I can code 2 plus

2 and get the value of 4 returned.

Just like in regular mathematics,

we use the plus symbol for addition.

Similarly, use the minus symbol to perform subtraction.

For example, 4 minus 2 returns the value of 2.

For multiplication use the asterisk or star

key so 4 times 4, will return 16.

For division use the forward-slash key

16 divided by 4 returns the value of 4.

You can use a double star sign to perform

exponential calculations for

slightly more complex arithmetic.

For example, 10 to the power of 2,

10 times 10 will return 100.

A final example is the remainder or the modulus operator.

This checks how many times you can fit one number

into the other and then return the remainder.

If I type 9 modulus 8,

the value of one is returned,

or 16 modulus 8 returns a zero.

This is because the number eight

divides into the number 16 evenly.

There is no remainder as represented here by the zero.

Besides basic mathematical operations,

you can also use parentheses to

group more complex calculations.

For example, 2 times 4 plus 8 can

return a different answer depending

on the order used for calculation.

For example, multiplying 2

times 4 and then adding the value of 8 returns

16 but 2 times the value of 4 plus 8 returns 24.

By using parentheses, I can

control the order of calculations.

Without parentheses, JavaScript will follow

the standard mathematical sequence of calculation.

In this case, the multiplication

would take precedence and be done first

so the 2 times 4 happens first and then

the 8 is added returning a value of 16.

Remember I can override that using

parentheses and get a different result.

That's an example of using parentheses and

JavaScript with numbers and mathematical operators.

Some of the most common JavaScript

mathematical operators are plus,

minus, times, division, and modulus.

If you'd like to learn more about mathematical operators,

there's an additional reading at the end of this lesson.

**Strings**

Since you're new to JavaScript,

you might not be familiar with strings.

Strings are used to represent and work with

a sequence of characters while programming in JavaScript.

Let me explain how they work.

To demonstrate the concept of strings,

I have the developer tools open in my browser and

the Console tab is selected. What is a string?

A string in JavaScript is a collection of

characters enclosed by single quotes, double quotes.

Such a collection of characters

is known as a string datatype.

Strings are one of the most common types

of data you'll be working with.

To code a string, I type in

a pair of single quotation marks.

This piece of code with just an opening and closing

single quote is referred to as an empty string.

To build an empty string,

I can also use double quotation marks.

While empty strings have their place in JavaScript,

they're not very versatile.

But strings don't have to be empty,

so let me build a non-empty string

with some letters and an exclamation mark.

I do this by typing the words hello there,

and an exclamation mark within

a pair of single quotation marks.

Besides letters and punctuation symbols,

we can add almost any other character

into a string, including numbers.

For example, I can type 'hello there! '

and follow it up with several symbols and numbers.

This is still a valid string.

There's one thing that strings can't

do and that is break onto the next line.

When I type a single quotation mark, the word hello,

and then press Enter before

typing the second quotation mark,

it results in an error which stops the code from running.

The same happens if I use double quotation marks.

When you use single or double quotes

to surround string values,

these characters are referred to as delimiters.

This is because I use them to delimit

a given string value from the rest of my code.

There are some unusual situations that can trip you

up when you first start using strings in JavaScript.

For example, let's say

that I want to create a new string.

I do this by typing a single quotation mark,

followed by the phrase,

it is a lovely day,

and I close the string with

a second single quotation mark.

Now I want to use the contraction of it is by

using an apostrophe but again, error code.

This is because JavaScript reads the word it,

that is the first two letters as forming a string,

and then it doesn't understand the rest

of the code because of the apostrophe.

To fix this issue,

I need to nest single quotes inside double quotes.

I do this by replacing

the original single quotation marks on

either side of the phrase with double quotation marks,

and now you'll notice there are no more errors.

You can make single and double quotes

too but it's better to stick with one or the

other because it can be confusing and lacks

consistency, and there you have it.

You have learned what a string datatype

is and what an empty string is.

You can also explain what

the string datatype is in JavaScript,

I encourage you to try and make

some more strings while practicing coding.

**Booleans**

The Boolean data type is used to

check if a statement is true or false,

which makes it a foundational part

of knowing how to use JavaScript.

It has many uses, but in this video,

I will focus on how to get the result of

a comparison to determine

if two values are the same or not.

What does a Boolean datatype?

To understand the answer to this question,

let's think about a real life scenario

of comparing numbers,

such as, for example,

checking if number 1 is smaller than number 2.

In JavaScript, I do it like this,

1 less than sign 2.

When I press the Enter key,

JavaScript says that this is a true claim.

Let's do some other comparisons.

Let's check if one is greater than two,

1 greater than sign 2.

This code returns a value of

false of the Boolean data type.

No matter how I compare numbers,

the result of the comparison will

always be either true or false.

There are no maybe's here.

That's why the Boolean data type has

only two possible values, true and false.

Besides the less than and the greater than operator,

I can also use some other operators to compare numbers.

It might help to think of

these various comparison operators, those tests.

Let's say I wanted to test if the comparison of

two numbers is correct or incorrect.

Now I will run a few tests

with some additional operators,

starting with checking for equality,

1 equals equals 2.

I'm claiming that 1 is equal to

2 and JavaScript returns false.

No, 1 is not equal to 2.

My claim is false.

When first introduced to

the syntax with two consecutive equal signs,

it is not uncommon for people to be slightly confused.

Why the double equals sign?

After all, you know that in mathematics you

only need a single equal sign to signal equality.

But in JavaScript, I use

a single equal sign as the assignment operator to assign

a value from the right side of the assignment operator

to the variable on

the left side of the assignment operator.

For example, I can type var score equals 100 to

declare a variable called score and

assign it to the value 100.

In other words, I can now confirm that the value of

score is 100 by typing out score in the console.

Indeed 100 it is.

In contrast, in JavaScript the equality operator

has to equal signs and it checks only for value.

Additionally, there's also

the strict equality operator that has

three equal signs and it checks for

both the value and type.

For example, if I use the equality operator,

let me type 100 double equal sign,

double quotation, number 100,

and close double quotation,

the console will return the Boolean value of true.

By using the double equals sign,

I compare only this value and not the type.

JavaScript checks only if 100 is equal

to 100 regardless of the datatype.

It ignores the fact that the 100 on the left of

the equality operator without

the double-quotes is a number,

and that the 100 on the right of

the equality operator with the double quotes is a string.

The strict equality operator is more unforgiving.

It checks for both the quality of

value and the quality of type.

Therefore, comparing the number of

100 without quotes with a string

100 with double-quotes will return

a Boolean value of false.

Why is this false?

It is false because although

the value is the same, the type is not.

The number 100 does not have

the same type as the string 100.

There's also an operator that's the opposite of

the equality operator, the inequality operator.

It's a combination of an exclamation mark

and an equal sign. Let's test it.

I take number 1,

exclamation mark the equals sign and number 1 again.

It returns a Boolean value of false because

it is false to say that one is not equal to one.

Additionally, just like there's

the strict equality operator,

there's also its opposite in

the form of the strict inequality operator.

This operator is an exclamation mark

with two equal signs.

Let's type number 1 as a number,

exclamation mark the equals sign twice,

and then number 1 inside

double quotes that is as a string.

This returns a Boolean value of

true because the number one is not

the same as a string 1 by both the value and the type.

It's true that these two values are not the same.

There are other operators in JavaScript,

and there are some additional concepts like

operator precedence and operator associativity,

but we'll skip this for now.

With the examples covered in this video,

you should have a basic grasp of how

the Boolean data type works and what it's useful for.

You will encounter Boolean values

again because there are other important ways

in which you can use them in

your code besides comparing numbers and strings.

**JavaScript improvements**

In this reading, you will learn about the history of JavaScript and the importance of ECMA (European Computer Manufacturers Association) and ECMAScript.

JavaScript is a programming language that had humble beginnings.

It was built in only 10 days in 1995 by a single person, Brendan Eich, who was tasked with building a simple scripting language to be used in version 2 of the Netscape browser. It was initially called LiveScript, but since the Java language was so popular at the time, the name was changed to JavaScript - although Java and JavaScript are in no way related.

For the first few years, after it was built, JavaScript was a simple scripting language to add mouseover effects and other interactivity. Those effects were being added to webpages using the *<script>* HTML element.

Inside each of the script elements, there could be some JavaScript code. Due to the rule that HTML, CSS, and JavaScript must be backward compatible, even the most advanced code written in JavaScript today ends up being written between those script tags.

Over the years, JavaScript grew ever more powerful, and in recent times, it's continually touted as among the top three commonly used languages.

In 1996 Netscape made a deal with the organization known as ECMA (European Computer Manufacturers Association) to draft the specification of the JavaScript language, and in 1997 the first edition of the ECMAScript specification was published.

ECMA publishes this specification as the ECMA-262 standard.

You can think of a standard as an agreed-upon way of how things should work. Thus, ECMA-262 is a standard that specifies how the JavaScript language should work.

There have been 12 ECMA-262 updates - the first one was in 1997.

JavaScript as a language is not a completely separate, stand-alone entity. It only exists as an implementation. This implementation is known as a JavaScript engine.

Traditionally, the only environment in which it was possible to run a JavaScript engine, was the browser. More specifically, a JavaScript engine was just another building block of the browser. It was there to help a browser accomplish its users' goal of utilizing the internet for work, research, and play.

So, when developers write JavaScript code, they are using it to interact with a JavaScript engine. Put differently, developers write JavaScript code so that they can "talk to" a JavaScript engine.

Additionally, the JavaScript engine itself comes with different ways to interact with various other parts of the browser. These are known as Browser APIs.

Thus, the code that you write in the JavaScript programming language allows you to: 1. Interact with the JavaScript engine inside of the browser 2. Interact with other browser functionality that exists outside of the JavaScript engine, but is still inside the browser.

Although traditionally it was possible to interact with the JavaScript engine only inside of the browser, this all changed in 2009, when Node.js was built by Ryan Dahl.

He came up with a way to use a JavaScript engine as a stand-alone entity. Suddenly, it was possible to use JavaScript outside of the browser, as a separate program on the command line, or as a server-side environment.

Today, JavaScript is ubiquitous and is running in browsers, on servers, actually, on any device that can run a JavaScript engine.

**Additional resources**

Here is a list of resources that may be helpful as you continue your learning journey.

T​hese resources provide some more in-depth information on the topics covered in this module.

[Mozilla Developer Network Expressions and Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators)

[Mozilla Developer Network Operator Precedence and Associativity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

[JavaScript Primitive Values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)

[ECMA262 Specification](https://tc39.es/ecma262/)

[jQuery Official Website](https://jquery.com/)

[React Official Website](https://reactjs.org/)

[StackOverflow Developer Survey 2021 Most Popular Technologies](https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies)

[Emojis](http://unicode.org/emoji/charts/full-emoji-list.html#1f600)

**Writing statements**

Every day from the moment we wake up,

you and I make decisions.

From what clothes to wear,

whether to have tea or coffee for

breakfast or even what movie to watch.

While some decisions are complex,

some can be relatively simple.

We make them based on whether

some condition is either true or false.

For example, if it's raining outside,

we will likely bring an umbrella.

The condition here is the weather.

If it's raining, we take an umbrella.

If it's not, we don't.

In this video, you will

learn about conditional statements in

JavaScript using the if and else statements.

In JavaScript, there are

pieces of code called conditionals,

and developers can use them to run code

conditionally based on whether

something is either true or false.

There are many types of conditional statements,

but the first one I want you to learn about

is something called the if statement.

The if statement checks a condition and will

execute a code block if the condition is met or true.

The condition criteria are essential here.

The block of code will only get executed if

the condition is evaluated to the Boolean value of true.

You can use many types of

comparison operators to evaluate the condition.

These include operators such as equal to,

greater than, less than,

and not equal to.

You can find a complete list of

comparison operators from the additional reading

at the end of this lesson.

So far, you've learned about using conditionals with

binary conditions where a thing

can be either true or false.

However, in the world of programming,

things are often not that simple.

Imagine you are a developer working on

a website where a person can

practice for their driving test.

Part of the driving test is that they must pass

a theory exam of 50 questions

based on the rules of the road.

The exam has only two results, pass or fail.

If the student's score is greater than 40, they pass.

If not, they fail.

You need to output the exam result to the learner.

This is where another conditional statement can be used,

and it's called the if-else statement.

This statement handles both outcomes of

our conditional request using the else keyword.

If a condition is true, do something.

If it's false, do something else.

Recall, a driving test example.

We can store the test result in a variable.

Suppose the variable value is greater than 40.

In that case, the true block is executed,

and you output a message to

the console telling the learner that they passed.

Suppose the variable value is less than 40.

In that case, the false block is executed,

and you output a message to

the console telling the learner that they did not pass.

There will be many occasions where you will need to

program for more complicated conditions or scenarios.

For example, you may want to test multiple conditions.

To do that, you can add

an additional block called the else-if statement.

In this video, you learned about

the conditional statements, if and else.

Conditional statements are a powerful tool,

and one of the fundamentals you need to

master when programming in JavaScript.

**Working with conditional statements**

Recall that conditional statements are used to run pieces

of code based on whether the condition is either true or false.

Okay, let me demonstrate this now with a simple example using a variable and

an, if else statement.

Additionally, we learn how to deal with cases where we have multiple,

if else, if conditions.

For example, the if else statement can be used when you want to execute

code based on a certain condition being either true or false.

But what if you have five conditions or even ten?

Well, you could use multiple lines of else If.

There is another way to do this, using something called a switch statement.

Okay, let me demonstrate this now with an example of when to use else if,

and when to use the switch statement.

So, I have VS code open here and

noticed that I have pasted in an example of an if else statement.

First, on line three here I've assigned the value of 50 to the variable result.

Then on line five I'm checking if the result Is greater than 40.

If this returns the Boolean value of true,

my code will console log the words you pass the test.

After that, on line seven, I specify the else block which will

run when the if condition returns false instead of true.

In that case, I use console.log to output, you did not pass the test.

Okay, so now let's run this code, and

as expected I get the output of you pass the test.

This happened because the statement 50 is greater than 40 in the if

condition returned true, and its associated block of code was executed.

Now, let me change the value of the result variable to say 30 and rerun the code.

This time I was notified that I did not pass the test.

This was because the condition 30 is greater than 40 was evaluated and

returned false.

The code was skipped inside the if block, instead,

the code inside the else block was run.

My code works well because I'm checking a binary condition here, but

what if there are more options?

In that case, I'll need to cover all these possibilities.

Let me switch tabs to demonstrate this using else if statements.

Okay, so notice that I have a variable named place and

it's assigned to a string with the value of first.

Next, notice my code contains four conditional statements.

The first, is an if statement that checks if the value of the variable

place equals the string value of first.

If this is true then a message of gold will output to the console.

As there are three more conditions, the next one needs to use an else if block.

This time I'm checking if the value stored in the place variable

equals the string value of second.

Once again, if this is true a message of silver will output to the console.

Notice lines nine and ten, this code is another else if block that checks for

a value of 3rd and outputs the message of bronze.

And this process would keep repeating for as many conditions as needed.

Finally, if no condition is met, I have a catch all else statement at the end.

This is crucial because if the conditional checks for the values first, second or

third are not satisfied, the else block will run and output the text, No medal.

Okay, so this code looks good to me.

Let me run it to test the output.

Notice the word gold is outputted, while this method is correct,

it can become a bit unmanageable when there are multiple conditions.

For example, say you had to evaluate ten conditions, luckily,

there is another statement at our disposal and this is called the switch statement.

Let me switch tabs again and now notice the code.

So, here I have a switch example, but I want to examine these files side by side.

So, let me click on the tab of the switch statement file and

I'll drag it around until I find a good place for it.

That will allow me to compare the code.

Let's adjust this a little bit more and clear the output of the console.

Okay, so you may notice that the syntax is significantly different

by Inspecting these two files.

If I run the switch statement code here I get the same result,

the word Gold is output.

Also, if I change the value of place variable to say fourth,

the default case will be run and the text, No medal, will output to the console.

The if else if, else example code acts precisely the same.

So, here when I update the variable place to fourth,

the same thing happens, the words, No medal are locked.

Congratulations, you have now learned about conditional statements using

the if and else.

You've also learned how to streamline multiple conditions using the switch

statement.

If you would like to learn more about conditional statements and operators,

there is a link to an additional reading at the end of this lesson.

**Conditional examples**

* In this reading, you will learn when to use the *if else* statement and when to use the *switch* statement.
* Both *if else* and *switch* are used to determine the program execution flow based on whether or not some conditions have been met.
* This is why they are sometimes referred to as **flow control statements**. In other words, they control the flow of execution of your code, so that some code can be skipped, while other code can be executed.
* At the heart of both flow control structures lies the evaluation of one or more conditions.
* Generally, *if else* is better suited if there is a binary choice in the condition.
* For example, in plain English: *if it's sunny, wear sunglasses. Otherwise, don't*.
* In this case, using an if statement is an obvious choice.
* When there are a smaller number of possible outcomes of truthy checks, it is still possible to use an *if else* statement, such as:

if(light == "green") {

    console.log("Drive")

} else if (light == "orange") {

    console.log("Get ready")

} else if (light == "red") {

    console.log("Dont' drive")

} else {

    console.log("The car is not green, orange, or red");

}

* However, if there are a lot of possible outcomes, it is best practice to use a switch statement because it is easier less verbose. Being easier to read, it is easier to follow the logic, and thus reduce cognitive load of reading multiple conditions.
* Nevertheless, this is not a rule set in stone. It is simply a stylistic choice.
* To reinforce this point, here's an example of the earlier *if else* conditional statement, using the switch syntax:

switch(light) {

   case 'green':

       console.log("Drive");

       break;

   case 'orange':

       console.log("Get ready");

       break;

   case 'red':

       console.log("Don't drive");

       break;

   default:

       console.log('The light is not green, orange, or red');

       break;

}

**Looping constructs**

In our everyday life,

there are times when we have to repeat

some activity again and again.

For example, closing each button when putting on a shirt,

washing a stack of plates one by one,

or counting down from ten to

one before singing Happy New Year.

Repetitive tasks are effective life,

and sometimes we also need to

do these in our JavaScript code.

In this video, you will learn about

looping constructs and how they can be

used to perform repetitive tasks.

Recall the conditional IF statement that

executes a block of code if a certain condition is met,

this sequence only runs once.

What if you needed to do this multiple times?

Well, to do this you need to do something called loops.

In JavaScript, developers use loops to continually

execute repeated blocks of code

until a certain condition is satisfied.

Loops are similar to conditionals and that a condition

must be satisfied in order for the code to execute.

But loops have an additional parameter,

which has many names.

But for the moment, you can think of this

as an incrementer or loop counter.

To understand why the counter variable is so important,

let's consider a real-life situation.

We're counting down from 10-1,

before singing Happy New Year.

In other words, only after we have completed the count,

we can start singing.

Those our countered needs to be a specific value,

a one and only after that,

we can sing a song.

In JavaScript, the counter can be any

variable or any number but in programming,

it's common to use the letter I.

The role of the counter is vital here, as, without it,

your loop will not know neither when to

start nor when to terminate,

and could run indefinitely.

This is known as an infinite loop.

For example, say you wanted to program

a count sequence like 1,2,3.

You could do this just by using the console.log method.

However, we are essentially

performing the same activity here.

The only difference is that we're changing

the number that is being logged to the console.

Using a loop, we could perform the same task

more efficiently and using less lines of code.

While this might seem like

a minimal gain for counting to three,

imagine counting to 100.

Using a loop, you can achieve this

with the exact same three lines of code.

You just have to update the condition.

There are many types of loops,

such as the for loop,

the while loop, and the nested loop.

You will learn about the nested loops

later in this lesson.

For now, let's explore the far end while loop.

First is the for-loop,

which is used to loop a block of

code a certain number of times.

The for-loop is a structured loop with

a specific conditional structure

where a counter variable must be set.

Then a condition must be specified and finally,

the counter must be incremented.

This can either be an increase or

a decrease depending on your needs.

The while loop is similar to the for-loop.

It will also run for as

long as the condition returns true.

However, the start counter is set outside of

the while loop and the incrementing

is done inside the loop's body.

Congratulations. In this video,

you learned about loop construction

using the far and while loop.

While you can achieve the same results using either a

for or while loop as a budding developer,

you might find it easier to work with the for-loop.

This is because the loop is

self-contained with all the loop's logic in one place.

Additionally, always remember

to update the counter value,

otherwise, the loop might loop forever.

**For loop**

Let's say you want to code a count from 1-3,

and then say go as if we're erasing.

You could do this by logging

each step to the console for 1,

2, 3 and go.

While this method works,

imagine a situation where you have to count from 1-10,

1-100, or even more,

where we have to type the console dialog method

a 100 times.

Luckily there's a more efficient way to do this

by using something called a for-loop.

Let me clear my counsel and paste

in an example of how to do this using a for-loop.

First is the declaration using the for keyword.

This declaration consists of

something known as the counter.

The counter is a variable which is

typically named i for iterator,

but you could use any variable name you want.

The counter has three conditions

for the for-loop to function.

First, the start count value must be assigned,

in this example it's one.

The second condition is the end-count value.

This specifies how many

times the loop will iterate or run.

In this example,

the condition is that the value of i must be

less than or equal to three for the loop to run.

The final part is the incrementor

which changes the value of i on each loop.

It's set to increase the number

stored in the variable i by one,

each time the code inside the loop runs.

You may have learned the technique i equal to i plus 1,

but you can also use the syntax i plus plus as

another way to increase the value of i by one.

If you do not specify how the value

of i should be increased after each loop,

the for loop will run forever,

as i would always be less than three.

You need something known as an exit-condition.

The exit-condition works with

the incrementor to prevent the loop from

running forever by specifying

at which value to terminate the loop.

In this example, the loop terminates when

the value of i is greater than three.

The loop will terminate when

the exit condition returns false.

In other words, until the value of i equals to 4.

Since 4 less than or greater than 3 returns false,

the for-loop will exit and JavaScript will

continue reading and running code below it.

The final part of the for-loop is

the loop-body which is enclosed in curly braces.

This contains the block of code that I

want to run on each loop-iteration.

In this example, I want to print the value of i

to the console using console-dialog method.

Notice that this code outputs

the same result as displayed earlier.

Well, let's see in an action.

Great, you've just created your first for loop.

There are different ways that you can work with for loop.

For example, I could start my count at 10

and set the value of i to be greater than zero.

Now, instead of incrementing

the value of i with plus plus,

I can decrement with the value minus minus.

This is essentially like counting backward.

Instead of saying go,

we might say Happy New Year.

I press, "Enter," and there is the result.

The countdown outputs from 10-1 and the greeting,

Happy New Year is displayed.

Let's break down what's happening here.

The console-dialog part of the for-loop was repeated

10 times until the value of i reached zero.

Once the value of i reached zero,

the condition i greater than 0

was no longer met and the loop exited.

Remember, a for-loop is a way to

automate repetition in JavaScript.

Why not try creating one in

your own code to repeat some code.

**While loop**

In this video you will learn about another type of loop,

the while loop and use it to code a countdown to the happy new year greeting.

The while loop is quite similar to the for loop but they're not exactly the same.

The first major difference is the counter value.

With a while loop this is set before the loop and must be clearly defined.

Let me demonstrate this now.

First, I create a variable called counter and set its value to 3.

Next I want you to know about the second major difference.

This is that you only need to specify the exit condition in the parentheses of

the loop declaration or put differently,

we just give JavaScript an expression to check.

If that expression returns true JavaScript will then run the code inside

the while loop.

In this example our condition will be counter is greater than 0.

Since the counter value of 3 Is greater than 0 it returns true.

So to construct the while loop I typed the word while and

then the condition inside the parentheses, this is counter greater than 0.

Next, inside the wild loops code block I need to tell it what to do.

In this example I want to run the console.log method to output the value

of the counter, then I need to decriminalize my counter by a value of 1.

This means that the first time JavaScript starts running the line of code with

the while loop declaration it checks the value of the counter variable.

If the counter value is greater than 0 and it is,

it will run the code inside the wild loop then it will output the current value of

the counter variable to the console which is the number 3.

Next, it will calculate the value of counter- 1, that is 3- 1 which is two.

It will then update the counter variable by assigning the number two as its

new value.

Finally it will return to the whiles condition and

check if the counter value is greater than 0, that is if two is greater than 0.

Since it is, it will go into the while loops code block again and

output the counters current value which is two.

Then it will update it again by assigning the result of 2- 1 as the updated

value of the counter variable.

Now since it's reached the end of the block, JavaScript will go back up and

confirm that it's true that 1 is greater than 0 then go inside the while loops

body again, output the value of 1 and update the counter to 1- 1.

Since it's reached the end of the loop's body, it will go back up and

check if 0 is greater than 0 and

since it isn't it will not run the code inside the while loop again.

Instead any code under the closing curly brace of the while loop will run.

To demonstrate this I add another line of code under this closing curly brace and

console log the string, Happy New Year and that's it.

Congratulations, you have now learned how to create and run a while group.

Why don't you give it a try?

Loops are a common way to automate code repetition.

If you would like to learn more about the specific use of loops there is

an additional reading at the end of this lesson.

**Nested loops**

What if you need your code to perform

more than one task at the same time?

Maybe you need to process data set A,

but at the same time also process data set B.

In cases like this,

you can use nested loops.

You might already be familiar with the for loop,

which is used for instructing your code to

perform the same task over and over again.

But with JavaScript, it's possible to nest loops within

other loops so that

multiple tasks can be performed at once.

There can be multiple levels of nesting too.

Let's now examine a practical example

where you might use a nested loop.

Suppose you are creating

a two-week plan for the days Monday to Friday,

you need to output

each day number and the week it's associated to.

You have five days for week 1 and five days for week 2.

Using a nested loop is

the perfect way to achieve this output.

Let me break it down for you now

using an example with for loops.

If we recall our example,

we have two categories; weeks and days.

The first loop, also known as the outer loop,

will loop over the weeks.

An increment counter of 2 is required,

and this is created with a variable named i.

Then the nested loop will loop over the days.

To do this, another increment counter is required,

and this is created with a variable named j.

Finally, the console log method is required to output

the text with the values of i and j to the console.

This code outputs 10 lines, week 1,

and each day from 1-5,

and then the same for week 2.

However, you need to take care not to overdo it

as multiple levels of

nested loops are not very performant.

In other words, the more nested loops there are,

the slower your code will run.

Let's now explore an example where you will output code

using nested loops to

display the summer months over two years.

I'm back in my browser's console and I

start by pasting in a for loop declaration.

Notice this specifies the year variable as 2023 and

the exit condition when the comparison of

year is less than 2025 returns false.

I've also set the year's value

to increase by one on each loop.

Essentially, I've coded this loop twice.

Next, I type console.log in

the loop body and place

the variable year inside the parenthesis.

I expect the output to display 2023 and 2024.

I run the for loop to confirm

that this is the case, and indeed,

the values 2023 and 2024 are logged to the console.

This is a process that you are already familiar with.

Now I want to demonstrate a nested loop,

and to do that,

I need to add another loop inside

the body of the first for loop.

This will also be a for loop.

This time I type for (var month equal 6,

month less than 9,

and month plus, plus).

What I'm doing here is counting only the summer months.

The summer months are usually June,

July, and August in the Northern Hemisphere.

That's months 6,

7, and 8.

Notice that the counter will start at month 6 and it will

keep checking if the numbers stored

in the month variable are less than 9.

On each subsequent loop,

the value stored in

the multi-variable will increase by 1.

Effectively, it's going to go through months 6,

7, and 8.

After it runs these three loops,

it's going to stop.

Now I type console.log in the body of the inner loop.

To make sure that I showed that this is nested,

I'm going to add several dash characters

and then the month variable.

Now, when I run this code,

the outer loop will run for the first time

and it will set the value of the year variable to 2023.

Then console log that number,

and then go into the code of the inner loop and run it.

Once inside the code of the inner loop,

I can expect the code to run three

times and output dashes 6,

dashes 7, and dashes 8.

The inner loop will then exit,

and the code will move back to

the outer loop to begin the next loop cycle.

When this starts, the value of the year variable

has been increased by one to 2024.

This still satisfies the condition

of year is less than 2025.

The code logs the year and proceeds to the inner loop.

JavaScript will go into the inner

for loop again and rerun it.

Once again, it will output 6,

7, and 8 again.

This code will continue this sequence display in

the year 2024 and the final three months of summer.

Let me run the code now.

Success. The code produced my predicted output.

Before I end this video,

let me demonstrate how to simplify the for loop.

Instead of using these long variable names,

I can just use i instead of the year,

and I can use j instead of the month.

This makes the code easier to read.

It might seem harder to understand because I'm using

i and j instead of more descriptive names.

But these variable names are standard practice in

the development community and are

considered a best practice to use.

You have now learned about

nested loops and the reason to use them.

It's important to remember that while

nested loops are an option for developers,

they can be resource-intensive.

Nested loops can be used in

instances where you need automated repetition of code,

for example, sorting numbers from highest to lowest.

A good developer writes code

efficiently to maximize resources,

and you are on your way to becoming one of them.

  [Week 1](https://www.coursera.org/learn/programming-with-javascript/home/week/1)

1. Uses of loops

**Uses of loops**

* In this reading, we'll discuss, at a very high level, the reasons to use loops in JavaScript.
* Note that we will keep this discussion high-level because there are multiple "pieces of the puzzle" that are still missing from your understanding at this point.
* This is why we will not get bogged-down in the detail of syntax and implementation, but instead, simply discuss how and why loops are used in everyday work of JavaScript developers.
* Consider the following example: You work as a developer for an online store.
* The store is selling letter cubes for toddlers, and the entire "Shop now" section of the site is organized in a layout where each cube on sale is displayed in a simple card component, with an image of the cube, the letter it teaches, a short description, and the price.
* Cards are organized in rows, so that each row contains three cards - three different letters.
* Each card is a preview of that specific letter cube on sale, and it's also a link to an entire page, dedicated to providing more info about the cubes, their teaching value, and providing the visitor with a way to complete their checkout process.
* Now, here's a quick question: where would loops fit into displaying this grid of cards showcasing the letter cubes on sale?
* To understand just how this works, let me code a basic prototype of how this might work.
* Since you still don't have enough knowledge to display website layouts in browser with the help of JavaScript, for now I'll have to settle for using a simple string and the console.
* Still, this should be a fun exercise.

var cubes = 'ABCDEFG';

for (var i = 0; i < cubes.length; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}

* ***Note: In order to have the styles applied, try running this code snippet in your browser's console.***
* That's it, with this simple code, the output in the console shows each letter on a separate line, styled like a letter cube for toddlers.
* The code itself should be mostly familiar, except for the *cubes.length* and the *cubes[i]* syntax.
* Without getting into too many details, here are both code snippets explained as simple as possible.
* The *cubes.length* returns a number. Since *cubes* is a string of characters the *cubes.length* gives me the length of the string saved in the variable.
* So this gives me the number 7, effectively making my for loop look like this:

for (var i = 0; i < 7; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}

* The second piece of code that's new here is the *cubes[i]* snippet.
* This simply targets each individual letter in the loop, based on the current value of the *i* variable.
* In other words, *cubes[i]*, when *i* is equal to *0*, is: *A*.
* Then, *cubes[i]*, when *i* is equal to *1*, is: *B*.
* This goes on for as many loops my for loop runs - and this is determined by the *cubes.length* value.
* It's also very versatile, since, if I, for example, decided to change the length of the *cubes* string, I would not have to update the condition of *i < cubes.length*, because it gets automatically updated when I change the length of the *cubes* string.
* There are some other ways to store data in JavaScript apps that you haven't heard about.
* But we can use the same approach with those other kinds of data, to achieve results that essentially work on the same principle as the one just described.
* Using loops is the essence of the approach taken in developing many different pieces of functionality in software today.

**Some additional examples**

* If I'm coding an email client, I will get some structured data about the emails to be displayed in the inbox, then I'll use a loop to actually display it in a nicely-formatted way.
* If I'm coding an e-commerce site selling cars, I will get a source of nicely-structured data on each of the cars, then loop over that data to display it on the screen.
* If I'm coding a calendar online, I'll loop over the data contained in each of the days to display a nicely-formatted calendar.
* There are many, many other examples of using loops in code.
* Using loops with data that is properly formatted for a given task is a crucial component of building software.
* In the lessons that follow, we'll learn about different ways of grouping related data and of displaying it on the screen using JavaScript.
* When combined with what you've already learned about loops, this gives you the skills to build various kinds of user interfaces where there is repetitive information.

Some more specific examples include:

* looping over blog post titles in some structured data, and displaying each blog post title on a blog home page
* looping over social media posts in some structured data, and displaying each social media post based on some conditions
* looping over some structured data on clothing available for sale in an online clothing store, and displaying relevant data for each item of clothing

Now you understand the importance of knowing how to work with loops in JavaScript. In the upcoming lessons, we'll learn other relevant information which will allow you to be able to do this.

**Additional resources for Conditionals and Loops**

Here is a list of resources that may be helpful as you continue your learning journey.

[Comparison Operators](https://www.javascripttutorial.net/javascript-comparison-operators/)

[Truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy)

[Falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy)

[Conditional statements](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals)

In JavaScript, there is also a shorthand version of writing a conditional statement, known as the conditional (ternary) operator: [Conditional (ternary) operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)

**Functions**

Have you ever wondered how when you turn on the dial or press a different button on

your washing machine for a specific wash, it just knows what to do.

You select sports wash and the machine runs a wash

cycle at the same temperature every time or you turn the dial to synthetics and

the machine runs a wash for the exact same length of time.

Each time functions work just like this, running the same set of stored

instructions repeatedly without requiring you to adjust the settings every time.

Over the next few minutes you learn about the advantages of using functions in

JavaScript, you will learn how functions are structured and

how to declare a function in your code with the use of parameters.

Finally, you will discover how to call or

invoke a function with your code with the use of arguments.

But first, what is a function and what does it do?

One of the basic principles of programming can be summed with

the acronym DRY in other words don't repeat yourself.

And it's thanks to functions that you can avoid repetition.

With functions you can take several lines of code that performs a set of

related actions and then group them together under a single label.

Then when you need to run the code that you've saved, you just invoke or

call the function.

You can run the code as many times as you want.

For example, suppose you want to add two numbers.

The process could be as follows.

First creates two variables a and b.

And assign each a number value, then create another variable c,

which holds the value of a and b added together.

Finally output the value of c to the console.

This piece of code works and adds two numbers together.

But what if you want to execute this code more than once?

Well you can do this by placing this block of code inside a function.

You declare a function by typing the keyword function, then space and

then the name of the function.

You can name the function nearly anything you want.

In this example it's named addTwoNums after the function name.

Place a left and right parenthesis.

Finally place a left and a right curly brace to complete the function.

The code to be executed will then be placed in between the left and

right curly braces.

And this is known as the body of the function.

It's important to remember that a function declaration on its own doesn't execute

the code.

It's just specifying what operations should be performed.

Once the function is executed to run the functions code you need to call or

invoke the function and this is achieved by typing the function named

followed by the opening and closing parenthesis.

Instead of saying run a function developers may say that they

call a function or invoke a function but don't worry.

All these terms mean the same thing to execute the code inside

the functions body.

Next let's explore another major advantage of functions.

The use of values.

Did you notice how the add to numbs example was just adding the numbers 10 and

20.

These numbers are are chosen values for this function but

this is just one example of the use of values with functions you can use

any values you want take for example the add to numbs function created earlier.

The function accepted, no arguments in other words,

no parameters were defined while variables can be used to produce results.

There are limitations as variables a and b have specific values and

the function would always give the same result.

So let's change this function to make it more flexible to do this.

You need to pass values to the function.

These values are known as function parameters and

are placed inside the function definition.

The values passed to the function are called arguments.

Let's explore this a little further now by setting a and

b as parameters instead of using variables.

I can easily plug in any two numbers that I want to add parameters

are sort of like variable names.

They allow us to set changeable values on each function call.

Functions with parameters can accept any kind of value and result in clean.

More efficient code building functions this way allows you to write flexible

functions that can be reused with various values as many times as you need.

For example by passing values as arguments to our function, addTwoNums.

We can call it as many times as we want and as long as we send it to values,

it will add them together and output the result.

In this lesson you learned about functions in JavaScript first you created

a function without parameters and

assigned fixed values inside the body of the function itself.

Then you learned about function parameters to pass values to functions.

It's important to remember that function parameters act sort of like variable

names.

The actual values that these variables will be assigned will come in the form of

function arguments.

At the time the function is invoked.

You're now familiar with the advantages of using functions declaring a function with

parameters and calling a function with arguments.

Working with functions is common in day to day development and

is one of the core skills a developer needs to master as they help you write

more efficient reusable code.

**Storing data in arrays**

You probably know that

freight trains are an important means for moving cargo.

But have you ever thought about

the system that keeps things running

smoothly as changes are made at every stop?

Carriages are typically set in a specific sequence,

and each carriage may carry different items or materials.

Other carriages can also be added or

removed without disrupting the rest of the order.

Much like a freight train,

JavaScript has a piece of functionality that allows you

to store and rearrange sequenced collections of items.

This is called an array.

In this video, I'll guide you through how to

build an array and access its contents.

I'll also demonstrate how JavaScript stores

the contents of the array and indexes its values.

Let's say that you want to emulate

a toy freight train using JavaScript.

On this train, each carriage has a number painted on

its site and is presented in sequence, starting from 0.

Using JavaScript,

you'll assign some cargo to each carriage.

For the first one, let's input

var carriage0 equals wheat.

Great. Now when you inspect the contents of carriage 0,

you'll get back the word wheat.

You can use the same approach to model your entire train.

Let's say you do just that and end

up with five variables of different values.

You've just built a train with five carriages,

each holding different cargo.

Now let's make things interesting and

introduce a second train.

This one only has two carriages with

the number 0 and 1 painted on the sides.

Seems pretty simple, right?

Well, you can't use the variable names

carriage0 and carriage1 because they're already taken.

You could use more descriptive names,

such as train2carriage0 and

a train2carriage1 but that feels a bit wordy.

Another problem is that you need to signal that

these variables are a collection belonging Train 2.

But when written this way,

JavaScript doesn't understand that.

Luckily, we can fix this.

To make JavaScript understand that we will group

a sequence of variables in a collection, we use arrays.

You can build an array using the array literal syntax,

which is an opening square bracket

followed by a closing one.

The array is currently empty,

but JavaScript now understands that

we would provide a collection of items.

Let's assign that collection to the Train 1 variable.

Finally, let's add carriages to the train 1

array by typing in

the item names with each separated by a comma.

Now we have a train.

But what happened to the carriage numbers? Not to worry.

Behind the scenes, JavaScript gives each item a number

starting from 0 that

corresponds to its place in the sequence.

You can then use these numbers to access each carriage.

For example, typing train1 followed by 0,

encased in brackets will

access the value inside carriage 0,

which is wheat in this case.

To summarize, arrays help us

achieve several things in JavaScript.

Arrays signal that the assigned values

all belong to a group.

These values are ordered in a sequence and each one

can be accessed through

its index number in that sequence.

In this video, you learned how to

build an array in JavaScript.

You also learned how

JavaScript sequences array items using an index value.

**Introduction to objects**

In programming, if you have

groups of data that you would like to relate,

you can assign them to something known as an object.

In this video, I'll help you to

understand why we use objects in JavaScript.

I'll also demonstrate how to build

objects using dot notation,

as well as by specifying comma delimited key-value pairs.

Let's say that you are using JavaScript to build

a game where the objective is

to build your own cookie selling business.

Your game is turn-based and

uses tiles for characters to move,

trade, and build their company.

Your first task is to build a store manager character.

You need to set certain traits,

so let's call them with JavaScript variables.

We'll set some numbers for the movement range,

social skills, street smarts, and help.

There. You've created a store manager,

but there are a few improvements

that you could make in your code.

One issue is the long length of the variable names.

Another is that your code does not explain to JavaScript

that these variables are related

and all describe the same character.

This is where objects come in.

Using objects, we can

resolve these pain points by shortening

our variable names and getting JavaScript to

understand that all those variables are related.

Let's start by declaring

the variable store manager to which you assign

an opening and a closing curly brace

This creates an empty object literal.

Next, instead of declaring

individual variables for the traits,

you can add a dot operator to each one,

right after the words store manager.

Now, when you inspect the store manager object,

it recognizes these as traits of the store manager.

Here you have built a store manager object

using dot notation.

Each trait is a property of the object.

Objects can be described as collections of

related properties where each property

is represented as a key value pair.

This means that what is normally

a variable name becomes a property key

and what is normally

a variable's value becomes

the property value of the object.

Objects can also be built by listing

the key value pairs inside of the object literal,

which specifies them as comma delimited properties.

To demonstrate, let's build a new character object.

First, I declare the object name

using var and then the object,

for example, assistant manager.

Then I simply input the properties and values in-between

curly braces with each property

followed by a colon and its value.

Next, I separate each key value pair by a comma.

Lastly, I finish the declaration

using a closed curly bracket.

That's it. I created an object and

assigned some values to its properties.

After an object is built,

you can still update

the object by adding new properties to it.

This can also be done with dot notation.

For the store manager object,

you can add another property called

next achievement by using dot notation.

As before, you do this by placing

a dot operator between the object name and the property.

Then for the assistant manager object,

you can add a property with the same name

using dot notation, and that's it.

Now your objects have

new properties added to the previous ones.

In this video, you've learned why objects are

valuable tools in JavaScript coding and

how to build them using dot notation and by

specifying comma delimited key value pairs.

Perhaps you can apply this knowledge

towards creating the next great gain.

**Object Literals and the Dot Notation**

By the end of this reading, you'll be able to:

* Explain one of the three common ways to build objects ( using the object literal notation)
* Outline the common way to add new properties to objects (or update existing properties) using the dot notation

**Object literals and the dot notation**

* One of the most common ways of building an object in JavaScript is using the object literal syntax: *{}*.
* To be able to access this object literal, it is very common to assign it to a variable, such as:

var user = {};

* Now an object literal is assigned to the variable *user*, which means that the object it is bound to can be extended and manipulated in a myriad of ways.
* Sometimes, an entire object can be immediately built, using the object literal syntax, by specifying the object's properties, delimited as key-value pairs, using syntax that was already covered in an earlier lesson item in this lesson.
* Here's one such previously built object:

var assistantManager = {

    rangeTilesPerTurn: 3,

    socialSkills: 30,

    streetSmarts: 30,

    health: 40,

    specialAbility: "young and ambitious",

    greeting: "Let's make some money"

}

* The beauty of this syntax is that it's so easily readable.
* It essentially consists of two steps:

1. Declaring a new variable and assigning an object literal to it - in other words, this: *var assistantManager = {}*

2. Assigning the values to each of the object's keys, using the assignment operator, *=*

Notice that it's very easy to build any kind of an object in JavaScript using this example syntax.

For example, here's a *table* object:

var table = {

    legs: 3,

    color: "brown",

    priceUSD: 100,

}

To access the *table* object, I can simply console log the entire object:

console.log(table);

The returned value is the entire *table* object:

{legs: 3, color: 'brown', priceUSD: 100}

Additionally, I can console log any individual property, like this:

console.log(table.color); // 'brown'

Now that I have this "syntax recipe", I can build any other object in a similar way:

var house = {

    rooms: 3,

    color: "brown",

    priceUSD: 10000,

}

An alternative approach of building objects is to first save an empty object literal to a variable, then use the dot notation to declare new properties on the fly, and use the assignment operator to add values to those properties; for example:

var house2 = {};

house2.rooms = 4;

house2.color = "pink";

house2.priceUSD = 12345;

Additionally, nothing is preventing me from combining the two approaches. For example:

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000}

house.windows = 10;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 10}

This flexbility additionally means that I can update already existing properties, not just add new ones:

house.windows = 11;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 11}

**Object Literals and the Brackets Notation**

By the end of this reading, you'll be able to:

* Explain how to build objects using the brackets notation
* Explain that with the brackets notation you can use the space character inside keys, since property keys are strings
* Explain that the keys inside the brackets notation are evaluated

**Object literals and the brackets notation**

* There is an alternative syntax to the dot notation I used up until this point.
* This alternative syntax is known as *the brackets notation*.
* To understand how it works, it's best to use an example, so I'll go through the process of coding the *house2* object again, in the same way that I did with the dot notation, only this time, I'll use the brackets notation.

var house2 = {};

house2["rooms"] = 4;

house2['color']= "pink";

house2["priceUSD"] = 12345;

console.log(house2); // {rooms: 4, color: 'pink', priceUSD: 12345}

* Note that using the brackets notation, I essentially just wrap each property's key **as a string**, inside either the single or double quotes - just like with regular strings.
* Then I wrap the entire property key into an opening and a closing square bracket.
* That's essentially all there is to it.
* I can both access and update properties on objects using either the dot notation, or the brackets notation, or a combination of both, like in the following example:

var car = {};

car.color = "red";

car["color"] = "green";

car["speed"] = 200;

car.speed = 100;

console.log(car); // {color: "green", speed: 100}

* For the time being, this is probably enough information on object creation.
* Before I discuss the topic of arrays and objects, let me just give you another important piece of information about the brackets notation.
* With the brackets notation, I can add space characters inside property names, like this:

car["number of doors"] = 4;

console.log(car); // {color: 'green', speed: 100, number of doors: 5}

Additionally, I can add numbers (as the string data type) as property keys:

car["2022"] = 1901;

console.log(car); // {2022: 1901, color: 'green', speed: 100, number of doors: 5}

* However, doing this is discouraged, due to obvious reasons of having a property key as a number string not really conveying a lot of useful information.
* Finally, there's one really useful thing that bracket notation has but is not available in the dot notation: It can evaluate expressions.
* To understand what that means, consider the following example:

var arrOfKeys = ['speed', 'altitude', 'color'];

var drone = {

    speed: 100,

    altitude: 200,

    color: "red"

}

for (var i = 0; i < arrOfKeys.length; i++) {

    console.log(drone[arrOfKeys[i]])

}

The above code will result in the following output:

100

200

red

* Using the fact that brackets notation can evaluate expressions, I accessed the *arrOfKeys[i]* property on the *drone* object.
* This value changed on each loop while the for loop was running.
* Specifically, the first time it ran, it was evaluated like this:
* The value of *i* was *0*
* The value of *arrOfKeys[i]* was *arrOfKeys[0]*, which was *"speed"*
* Thus, *drone[arrOfKeys[i]]* was evaluated to *drone["speed"]* which is equal to *100*

This allowed me to loop over each of the values stored inside the *drone* object, based on each of its properties' keys.

**Arrays are Objects**

By the end of this reading, you'll be able to:

* Explain that arrays are objects, with their own built-in properties and methods
* Outline the common way to extend arrays using the push() method
* and explain how to trim the last member of an array using the pop() method

**Arrays are objects**

In JavaScript, arrays are objects. That means that arrays also have some built-in properties and methods.

One of the most commonly used built-in methods on arrays are the *push()* and the *pop()* methods.

To add new items to an array, I can use the *push()* method:

var fruits = [];

fruits.push("apple"); // ['apple']

fruits.push('pear'); // ['apple', 'pear']

To remove the last item from an array, I can use the *pop()* method:

fruits.pop();

console.log(fruits); // ['apple']

Tying into some earlier lessons in this course, I can now build a function that takes all its arguments and pushes them into an array, like this:

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(one);

    arr.push(two);

    arr.push(three);

    console.log(arr);

}

I can now call the *arrayBuilder()* function, for example, like this:

arrayBuilder('apple', 'pear', 'plum'); // ['apple', 'pear', 'plum']

Even better, I don't have to console log the newly built array.

Instead, I can return it:

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(one);

    arr.push(two);

    arr.push(three);

    return arr;

}

Additionally, I can save this function call to a variable.

I can name it anything, but this time I'll use the name: *simpleArr*.

var simpleArr = arrayBuilder('apple', 'pear', 'plum');

And now I can console log the values stored in *simpleArr*:

console.log(simpleArr); // ['apple','pear','plum']

**Math object cheat sheet**

JavaScript has handy built-in objects. One of these popular built-in objects is the Math object.

By the end of this reading, you'll be able to:

* Outline the built-in properties and methods of the Math object

**Number constants**

Here are some of the built-in number constants that exist on the Math object:

* The PI number: *Math.PI*
* The Euler's constant: *Math.E*
* The natural logarithm of 2: *Math.LN2*

**Rounding methods**

These include:

* *Math.ceil()* - rounds up to the closest integer
* *Math.floor()* - rounds down to the closest integer
* *Math.round()* - rounds up to the closest integer if the decimal is *.5* or above; otherwise, rounds down to the closest integer
* *Math.trunc()* - trims the decimal, leaving only the integer

**Arithmetic and calculus methods**

Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the *Math* object:

* *Math.pow(2,3)* - calculates the number *2* to the power of *3*, the result is *8*
* *Math.sqrt(16)* - calculates the square root of *16*, the result is *4*
* *Math.cbrt(8)* - finds the cube root of *8*, the result is *2*
* *Math.abs(-10)* - returns the absolute value, the result is *10*
* Logarithmic methods: *Math.log()*, *Math.log2()*, *Math.log10()*
* Return the minimum and maximum values of all the inputs: *Math.min(9,8,7)* returns *7*, *Math.max(9,8,7)* returns *9*.
* Trigonometric methods: *Math.sin()*, *Math.cos()*, *Math.tan()*, etc.

**Math object**

In this video, you'll learn how to use the random and

seal methods of the math object in JavaScript.

The math object is really useful when using some of its built in methods,

one of which is the random method.

For example,

this method allows you to generate a decimal number between 0 and 0.99.

So, let's explore how the random method works in practice.

One way you can work with the math.random method,

is to save its result in a variable.

Notice I have created a variable named decimal and

assign its value to the math.random method.

I can now log the value to the console, but

please note the limited range of numbers that will be returned.

If you want to output a number larger than one, you must multiply the variable by 10.

And I can do this with the multiplication operator using the asterisk symbol.

When I click on the run button, the output shows two numbers.

The first one is a random decimal, and

the second one is the decimal multiplied by 10.

So, this is an example of how you can extend the range of the math random method

to meet your needs.

Okay, let me switch tabs now and

learn about another built in method of the math object.

The Ceil method.

This is used to round up any decimal number to the nearest integer.

Let's pass several different decimals through this method and

inspect the output.

Notice that I have some code pasted into this file.

And online 4, I've created a variable named rounded and

assigned it to the math.ceil method.

Inside the parentheses, I have placed the value 0.0001.

I've logged this variable to the console.

When I run my code notice that a value of one is output.

This is because the seal method can only round upwards.

Just to confirm that behavior, we can change the decimal.

When I set it to 0.5, I get the same outcome.

I change it to 0.99 and as expected, 1 again.

Okay, let's move on to working with some larger numbers.

Let me in comment line seven and notice that 1.01 returns a value of 2.

When the value is 1.5, 2 is also of the output.

Finally, if I run the seal method on the value 2.99 I end up with 3.

So, now you understand how both the random and seal methods work.

Let me switch tabs again and next build some code that will combine the two to

return a random integer between 0 and 10.

The first step is to create a variable and

assign it the value of the math.random method multiplied by 10.

I need to pass this variable through the ceil method to ensure that an integer

instead of a decimal value is created.

So, I declare the variable rounded and assign it the value of the math.ceil

method containing the decimal variable inside the parentheses.

Finally, I use the console.log method to output the variable rounded.

If I run the code, notice that a value between 0 and 10 is output to the console.

Just to confirm this works, I'll rerun the code and

once again I get the expected results.

In this video, you have learned how to use the random and

seal methods of the math object in JavaScript.

Can you think of editing creative uses for these?

Why not give them a try in your own code?

**A closer look at strings**

When two JavaScript developers

discuss a subject related to work,

sometimes you might hear

the word iterable being mentioned.

But what exactly does iterable mean?

Well, in JavaScript an iterable is

any datatype that can be iterated

over using a for of loop.

Some of the data types you can iterate

over are arrays and strings.

In this video, you will learn about

the array characteristics of strings,

and you will also explore

further string manipulation

using the concatenation operator.

In the world of JavaScript,

it can often be said that strings behave

like arrays as strings are array-like.

For example, you can run a for loop over an array of

letters and output the result

using the console.log method.

Additionally, you can also run a for loop over a string.

For example, loop over the string A,

B, C to output the letters individually.

Let me demonstrate this now with some code examples.

In the file I have opened in my console,

I have a variable veggies,

which has an array with three strings.

Each string holds the name of a vegetable,

specifically onion, parsley, and carrot.

I've typed a console log for veggies.length,

which will return the number of array items.

I also have console logs for the first and second item of

the array using the indexes zero and one respectively.

Finally, I have an example of a for

loop that sets the counter of i to zero.

The exit condition for this loop is

that the comparison of i is less than

veggies length returns false and i

plus plus is the value to increment on each loop.

Inside the for loop,

I'm console logging the veggies at the index of i.

When I run this code,

I get a three to represent the array length.

Onion from index 0,

parsley from index 1,

and then finally onion,

parsley and carrot as a result of the

for loop outputting all the items in the array.

Now, let's clear

the console and move on to the next file.

Here I'll explore how strings can be iterated over.

In this file, I have

a variable named greeting with a string value of howdy,

along with console logs for the string length,

and the characters at indexes zero and one.

Again, I have a for loop that is

structurally the same as the previous example.

Running this code gives us five for

the string length H and O for

the zero and one indexes and then each of

the characters of the full string in

sequence as an output of the loop.

Despite the similar behavior,

strings are not arrays.

To examine this more closely,

let's move on to the next example.

Let me clear the output first.

In this code, I have the greet

variable that stores a string

hello and a user variable that stores a string Lisa.

Then I run a console log with

the string greet and the pop method,

which as you know,

removes items from an array.

There are two other lines of code,

but don't worry about them now,

I'll cover them shortly.

When I try to run this code,

I get an error that says type

error greet.pop is not a function.

This tells me that I can't run

all the array methods on strings.

Now, let's run the two other lines of code.

On the first one, I console log the variable greet,

the plus operator, and the variable user.

On the next line, I console log

the variable greet and

concat method with the variable user.

The concat method accepts whatever value I

want to concat or join with the greet variable.

In this case, both lines of code will work.

They both console log Hello Lisa.

I can confirm that the plus operator here,

when used on strings,

acts as a concatenation operator,

meaning it join strings together.

I also have a method named concat that I can use on

a string to add values to whatever

string the concat method was running.

**String cheat sheet**

By the end of this reading, you'll be able to:

* Identify examples of String functions and explain how to call them

In this cheat sheet, I'll list some of the most common and most useful properties and methods available on strings.

For all the examples, I'll be using either one or both of the following variables:

var greet = "Hello, ";

var place = "World"

Note that whatever string properties and methods I demo in the following examples, I could have ran it on those strings directly, without saving them to a variable such as the ones I named *greet* and *place*.

In some of the examples that follow, for the sake of clarity, instead of using a variable name, I'll use the string itself.

All strings have at their disposal several built-in properties, but there's a single property that is really useful: the *length* property, which is used like this:

greet.length; // 7

To read each individual character at a specific index in a string, starting from zero, I can use the *charAt()* method:

greet.charAt(0); // 'H'

The *concat()* method joins two strings:

"Wo".concat("rl").concat("d"); // 'World'

The *indexOf* returns the location of the first position that matches a character:

"ho-ho-ho".indexOf('h'); // 0

"ho-ho-ho".indexOf('o'); // 1

"ho-ho-ho".indexOf('-'); // 2

The *lastIndexOf* finds the last match, otherwise it works the same as *indexOf*.

The *split* method chops up the string into an array of sub-strings:

"ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']

There are also some methods to change the casing of strings. For example:

greet.toUpperCase(); // "HELLO, "

greet.toLowerCase(); // "hello, "

Here's a list of all the methods covered in this cheat sheet:

* *charAt()*
* *concat()*
* *indexOf()*
* *lastIndexOf()*
* *split()*
* *toUpperCase()*
* *toLowerCase()*

**Object Methods**

* You might already be familiar with objects in JavaScript.​
* In this video, you will learn how to design objects as combinations of data and functionality.
* As you might already know, an object consists of key-value pairs, known as properties.​
* We can add new key-value pairs to existing objects using the dot notation and the assignment operator. ​

var car = {};

car.color = "red";

These are known as properties, and can take many data types, including functions.

var car = {};​

car.color = "red";​

car.turnKey = function() { ​

  console.log('engine running'); ​

}

If the function is a property of an object, it is then referred to as a method. ​

This is a function that can be accessed only through the JavaScript object that it is a member of. For example, the log method, which belongs to the console object, can only be accessed through the console object.​

*console.log('Hello world'); ​*

Let's explore this further now. I will create an object using something known as the constructor function.

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

* First, I'll build a new object literally named *car*. I type *var*, space, *car*, space, equal sign, space, followed by a set of curly braces, and finally a semicolon.
* Now, I'll extend the *car* object by assigning it a property named *mileage*. ​
* When I inspect the object, I can confirm that it contains a *mileage* property set to *98765*. ​
* I want to add another property to the *car* object. This time, I will add a property named *color* and set it to the value of *"red"*.
* I can inspect the object again by typing its name into the browser console. So now, when I type *console.log(car)*, I get an object with two properties: the *mileage* property, which is set to *98765*, and the *color* property, set to *"red"*. ​
* Great, now I've added two properties to my object.
* Next, I want to add a method to my *car* object. And this method, when called, will output some text to the console. ​
* So, once again, I add another property to my *car* object. After all, a method is just another property of the *car* object. It's just another key-value par that the car object holds.
* What's unique is that the value I'm assigning to it is a function. ​

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

console.log(car);

{ mileage: 98765, color: 'red' }

{ mileage: 98765, color: 'red', turnTheKey: [Function (anonymous)] }

* So, I begin by typing *car*.*turnTheKey*, equals, and then I type the code for my function. So *function*, open-close parentheses. Then the two curly braces where I will place my code. Finally, inside the curly braces, I type the console.log followed by the message *"The engine is running"*.
* Now I can inspect my *car* object again by typing its name into the console log method. This time, it displays that the *car* object contains three properties; the *color* property, the *mileage* property, and the *turnTheKey* property. ​
* Remember that all the key-value pairs in an object are referred to simply as properties. However, if I want to differentiate between the properties that can be executed, then I refer to such properties as methods. ​
* So, now I want to add another method to the *car* object. I'll name this one *lightsOn*. ​
* Once again, I type *car.lightsOn*, and then I add an equals sign, and again since it's a method, I'm assigning it to a function. This function will also have a console log in its body, and I'm just logging the string with the text *"The lights are on"*. ​

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

car.lightsOn = function() {

    console.log("The lights are on.")

}

console.log(car);

car.turnTheKey();

car.lightsOn()

{ mileage: 98765, color: 'red' }

{

mileage: 98765,

color: 'red',

turnTheKey: [Function (anonymous)],

lightsOn: [Function (anonymous)]

}

The engine is running

The lights are on.

* Ok, so now I have added four properties to my object. And two of those are methods. ​
* I've already ensured that I'm getting the correct *mileage* and *color* from my *car* object. Now, I'll try executing the *turnTheKey* and the *lightsOn* methods. ​
* First, I'll invoke the *turnTheKey* method. ​
* Remember that this method can be accessed only through the *car* object, so I first need to type the name of the object that holds the *turnTheKey* method. In other words, I need to type the word *car*, followed by a dot, and then the name of my method, which is *turnTheKey*. ​
* Remember that this property is a method. So, to run it, I need to append an opening and a closing parenthesis so that the JavaScript engine can process my JavaScript code. ​
* Notice that this results in the *"The engine is running"* string logged to the console. ​
* Now I'll test the other method. Once again, I need to access it through the *car* object, so I type *car.lightsOn*, and again, I need to add those parentheses to invoke the *lightsOn* method. I press the ENTER key and notice the text displays in the console.
* Success! It's important to remember that when the JavaScript engine runs this line of code, it locates the *car* object in its memory. Then, it finds the *lightsOn* method on the *car* object. It reads the function declaration that's saved on this property and runs it, line by line.
* Since there's only a single line of code, the JavaScript engine logs the string *"The lights are on"* to the console. ​

**Typeof**

In this video, you'll learn how to use the type of operator to identify the data type

of elements in JavaScript.

The type of operator accepts and evaluates a parameter and

returns the name of the data type represented as a string.

To use it, you can type typeof followed by the parameter enclosed within parentheses.

In my code, notice that I have created several instances of the variable test.

Each is assigned the type of operator with a different parameter.

To check the result,

I'm also using the console log method to output the value of the variable test.

Okay, so as you might notice I have commented out all the lines of code

containing the test variables.

To demonstrate the type of operator using different parameters.

I will uncommon each of these lines one at a time.

Run the code and then inspect the output.

First., uncommon line 3 which contains the type of operator passed

a parameter of a string.

When I run the code,

notice that the name of the data type is output which is a string.

Next, let me uncommon line 5 which contains a number value.

I rerun the code and notice that the data type named number is output.

If I perform these steps on a decimal number I get the same result for

the decimal number 3.14.

Okay, let's now check the Boolean data types of true and false.

If I run the code the data type named Boolean is output to the console.

I uncommon line 11, rerun the code and notice that Boolean is also output.

Okay, next let me show you that you can also pass comparisons as they will

evaluate to a single value.

In essence a single parameter.

For example, you can pass a condition like 1<2.

When I run the code, Boolean is output.

You can also pass an array as a parameter to the type of operator.

However, the result might be somewhat surprising.

For example, here I have an array containing the numbers 1, 2 and 3.

If I run the code, the data type named object is output.

This is because arrays in JavaScript are actually objects.

Passing an object through the type of operator also returns object.

Finally, you can pass a function as a parameter of the type of operator.

If I uncommon line 19 and run the code.

Notice that the data type name function is output.

Okay, and that's it for this video.

You now know about the type of operator and

how it can be used in JavaScript to check the data type.

I encourage you to try it out in your own practice code.

**Additional resources**

Here is a list of resources that may be helpful as you continue your learning journey.

[JavaScript Functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)

[JavaScript Object Basics](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics)

[typeof operator in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof)

[Arrays are "list-like objects"](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays)

**Bugs and errors**

Isn't it frustrating when you're

working at your computer and

an error message appears on

the screen and forces you to stop what you're doing.

When a bug happens,

our program keeps running,

but it behaves in a way we didn't intend.

When an error happens,

our program stops running as the result of an error.

In this video, you will learn about bugs and

errors in JavaScript and some of the common error types,

such as syntax, reference and type errors.

Suppose you have created

a function that you pass two numbers.

The function adds the two numbers and outputs the result.

Next, you call the function and

add another console log to the next line,

where you will output the string still running.

When you run the code,

the number 3 will output.

Then JavaScript will happily continue onto

the next line of code and

console log the string still running.

This is because our example code contains no errors.

When the function is called,

the code is executed and then moved to the next line.

While this code works as expected,

what if you pass to the function some varying data types,

such as a string and a number?

For example, a string of one and number of two.

For this, recall how concatenation works.

Due to JavaScript coercion,

when you add a string and numbers

using concatenation with the plus operator,

the values will concatenate as strings.

Thus, the result is a string of 12.

Our function is now no longer adding numbers.

Instead, it is concatenating strings.

However, JavaScript happily continues

executing the code,

outputs the string of 12 and

console logs the words still running.

Notice that our program is working without interruption.

However, it is not working how we intended it to work.

This is what's referred to as a bug,

as it makes our code behave unexpectedly,

but it continues to work.

How does an error differ from a bug?

Well, let's explore another scenario.

Suppose you had simpler programming

that has two statements.

The first uses console log to output c plus d,

and the second one uses

console log to output the statement,

this line never runs.

What at these two lines of code

were the only lines of code in our program.

Where would JavaScript find

the values of the variables c and d?

Well, it wouldn't because in the sample code,

these variables don't exist.

They have neither being declared or assigned.

It would look for the variables of

c and d. Since they don't exist,

JavaScript outputs a reference error.

This type of error occurs when a value is not defined,

but you attempt to use it in your code.

When an error happens,

our program stops execution

the code as a direct consequence.

No further lines of code are executed.

In JavaScript and some other programming languages,

we say that an error is thrown.

An error can be defined as a faulty piece of

code that prevents the program from further execution,

an error gets thrown and the program stops.

JavaScript does its best with reporting error

by outputting an error message to the console.

This error reporting is useful

because it narrows down the issue with our code.

In JavaScript, there are many error types

and some of the most common are syntax error,

type error, and reference error.

You have just learned about reference error.

Let's explore syntax error and type error briefly now.

A syntax error occurs when you write

a piece of code that JavaScript cannot read.

For example, if you declare

a variable and assign it a value of a string,

but forget one of the closing quotation symbols.

Alternatively, suppose you try

to run an array method on a number.

For example, the pop method.

In this example, a type error

would be reported on called type error,

5.pop is not a function.

You've now learned about bugs and

errors and the differences between them.

You've also learned about some of

the common errors and how

JavaScript helps you catch these

by outputting error messages.

It's a fact that programming errors will

almost certainly occur every time you write code.

Next, you'll learn about some of

the different JavaScript tools

used to catch these errors.

**Try catch blocks**

Have you ever watched two people playing a game of catch?

It's a pretty simple game.

One player throws a ball and the other tries to catch it.

You may recall that when your code contains an error, it stops running.

Well, in JavaScript, there are some built in statements to help your

code continue to run even if an error occurs.

They also use keywords like throw and catch.

However they tried to catch the error instead of the ball.

In this video, you will learn about the statements throw, try and

catch and how they can be used to work with errors in JavaScript and

prevent your code from stopping.

This process is more commonly known as error handling.

First, let's explore the try,

catch statement that uses the key words, try and catch.

If a piece of code throws an error, it can get wrapped inside a try block.

Then you can catch the error with the catch block, and use it to do something.

For example, output the error message to the console.

Another key word that you need to be aware of is that throw keyword.

Using the throw keyword,

you can force an error to be thrown from the try block to the catch block.

It's important to remember that you can use the throw

keyword outside the try block, but it will not be possible to catch it.

The catch block accepts something called an error which is an object.

This is the actual error that is thrown from the try block.

While you can name it anything you like, it's best to keep it short and meaningful.

in this case I named it Err.

Let me demonstrate how this works further with some code examples.

Let's define a block of code to be tested for errors using the try catch statement.

Okay, so I have visual studio code open and

a JavaScript file with some sample code.

If I was to run this code now it would return an error and stop working.

This is because the first line uses a console log to output a+b.

But these variables are not declared anywhere.

The second line has a console log to output the string.

This line is never reached and this statement looks to be or free.

When I click the run button, a reference error is output,

telling me that a is not defined.

The JavaScript engine then stops immediately and

does not process the 2nd line.

It's not only JavaScript that throws errors in our programs,

you can actually throw them on purpose.

This is done by typing the keyword throw followed by the keyword new and

then a specific errors constructor.

In this case I use the reference error constructor with a pair of parenthesis at

the end.

When I run the code this time I get a reference error output again.

So how can we prevent errors from stopping our programs?

This is where the try catch syntax makes itself useful.

Let's go to a separate file with code that shows how that works.

But first let's break down the structure of the code.

The try block starts with the try keyword.

And inside of curly braces, you place the code that you think will throw an error.

Next is the catch block which catches the error that the try block produces.

It begins with the catch keyword and in parenthesis you have a built in or

object that you can name whatever you like.

But here I've used err.

Inside the curly braces, you place the code you would like to execute.

In this example,

the try block contains the same console log statement as before to output a+b.

The catch block contains console logs to output two strings.

There was an error and the error was saved in the error log.

Finally, after the catch block, another console log outputs the string,

my program does not stop.

The benefit of using try catch is that even if JavaScript throws an error while

going through our code, it will not stop program execution.

To demonstrate that let's run our code example.

Notice again a reference error is output stating that a is not defined, and

then the two strings from our catch block.

It's important to understand here that the error is output because I logged it to

the console.

The strings output after the error message proving our program continued running.

Finally let me switch tabs to another file, and

demonstrate how JavaScript responds when I manually throw an error.

In this example,

the try block contains the throw statement to throw a new reference error.

In the catch block there are two console logs.

The first one outputs the error object and

the second outputs the string, there was a reference error.

Outside the try catch blocks, the program ends with a console logs to

output the string, My program does not stop.

When I run this code, notice that the reference error is output.

This is the error that was thrown in the try block and

then output using the error object in the catch block.

Once again, this is just the error being output.

The code has not stopped running.

And I can confirm this as the final console log message is output to

the console.

In this video, you've learned how to work with errors in JavaScript.

You learned how you can use try and catch blocks and

the throw keyword to deal with them.

The next time you see a game of catch, think about how you can use a similar

mechanism to help catch errors in your code.

**Syntax, logical and runtime errors**

By the end of this reading, you'll be able to:

* Recognize common types of errors in JavaScript

Here are some of the most common errors in JavaScript:

* ReferenceError
* SyntaxError
* TypeError
* RangeError

There are some other errors in JavaScript. These other errors include:

* AggregateError
* Error
* InternalError
* URIError

However, in this reading I'll focus on the Reference, Syntax, Type, and Range errors.

**ReferenceError**

A ReferenceError gets thrown when, for example, one tries to use variables that haven't been declared anywhere.

An example can be, say, attempting to console log a variable that doesn't exist:

console.log(username);

If the variable named *username* hasn't been declared, the above line of code will result in the following output:

Uncaught ReferenceError: username is not defined

**SyntaxError**

Any kind of invalid JavaScript code will cause a SyntaxError.

For example:

var a "there's no assignment operator here";

The above line of code will throw the following error:

Uncaught SyntaxError: Unexpected string

**There's an interesting caveat regarding the SyntaxError in JavaScript: it cannot be caught using the *try-catch* block.**

**TypeError**

A TypeError is thrown when, for example, trying to run a method on a non-supported data type.

A simple example is attempting to run the *pop()* method on a string:

"hello".pop() // Uncaught TypeError: "hello".pop is not a function

The array-like behavior of strings was already covered in an earlier lesson in this course.

However, as can be confirmed by running the above line of code, strings do not have all the array methods readily available to them, and trying to use some of those methods will result in a TypeError being thrown.

**RangeError**

A RangeError is thrown when we're giving a value to a function, but that value is out of the allowed range of acceptable input values.

Here's a simple example of converting an everyday *Base 10 number* (a number of the common decimal system) to a *Base 2 number* (i.e binary number).

For example:

(10).toString(2); // '1010'

The value of *2* when passed to the *toString()* method, is like saying to JavaScript: "convert the value of *10* of the Base 10 number system, to its counter-part in the Base 2 number system".

JavaScript obliges and "translates" the "regular" number 10 to its binary counter-part.

Besides using Base 2 number system, I can also use the Base 8, like this:

(10).toString(8); // 12

I get back the value *12*, which is the plain number 10, writen in Base 8 number system.

However, if I try to use a non-existing number system, such as an imaginary *Base 100*, since this number system effectively doesn't exist in JavaScript, I will get the RangeError, because a non-existing *Base 100* system is **out of range** of the number systems that are available to the *toString()* method:

(10).toString(100); // Uncaught RangeError: toString() radix argument must be between 2 and 36

In this reading, you've covered some of the most common errors in JavaScript.

**Undefined, null and empty values**

Imagine that a restaurant has

been booked for a wedding reception,

and the big day has finally arrived.

Unfortunately, not everything is going smoothly and it

seems that many expected guests have not yet arrived.

How do we know that empty seats

should be filled with guests?

It's as simple as placing a small sign on each table.

Each one tells us a person is not currently here,

but a person should be here.

In JavaScript, there are similar situations in which you

need to mark the places where

a value or object should exist.

In this video, you'll learn about three types

of empty values: the null datatype,

the undefined datatype, and empty strings.

I'll also demonstrate some of the scenarios in

which you can expect to encounter these values.

We'll start with the null datatype.

Null represents the intentional absence

of any object value.

It is also a return value

of some built-in JavaScript methods.

For example, let's say you create

the variable named letters and

assign it a string value of abc.

You'd like to search for the letter a in the string.

You could do this by using

the match method to search inside the variable.

This returns an array with

several pieces of information,

but the most important part is that it confirms

the a was found in the string.

This time, let's use

the same method on a different letter,

the letter d. The return value

of the match method should be an array,

which is a object in JavaScript,

but since the d letter cannot be found,

the array with the result can't be built,

so null appears instead

to indicate the absence of an object.

In JavaScript, there may also

be times when you are building something that

hasn't been clearly defined

yet and so you can't assign a value to it.

Fortunately, there is a way to store it so that you

can assign it later using the undefined datatype.

While some data types in

JavaScript can hold many possible values,

others are constrained to just a few.

For example, the string data type

can hold a virtually infinite combination of

characters while the Boolean data type

is limited to the values true or false.

The undefined data type can only

hold one value, undefined.

You may recall seeing this in your practice code.

For example, all functions return undefined by

default unless it's been decided

to return a specific value instead.

When you use console.log to output something like a name,

you will see undefined displayed after the output.

This is because console.log is

a function and you are not returning a value.

Another common situation where the undefined value

appears is when a variable

is declared without an assignment.

For example, suppose I create a variable named noise,

but do not assign it a value.

You might recall that this is valid

JavaScript and the code will run.

However, as the variable has not been assigned a value,

JavaScript automatically assigns the value of undefined.

We can even explore this further.

For example, let's say you use console.

to output the unassigned variable noise.

Then on the line below,

you assign the noise variable

the string value of thunder.

What will happen if you then output

this variable again to the console?

The console will output the string thunder.

This means only the instances after

the declaration was take the assigned value.

Even if the declaration assigns a value,

any uses of the variable before

the declaration will still return undefined.

While it's important to understand this behavior,

it's usually best practice to always assign

your variables with values when you declare them.

As you may have noticed,

JavaScript gives you some flexibility,

but you still have limitations.

For example, if you tried to console.log

a variable that hasn't been declared in your entire code,

your program execution will stop,

and rather than undefined,

a reference error will be output.

One way to think about it is that

the undefined datatype acts like

a placeholder for a value that

the JavaScript engine knows to exist.

It just has not been specified.

Another scenario of undefined is when you

try to access an object property that doesn't exist.

For example, let's say you have

a game object with

a property score that has a value of 1,000.

The property contains all lowercase words.

Let's say you try to access this property,

but you make a spelling mistake and spell score with

a capital S. JavaScript

can't find anything with this information,

so it gives you the value undefined instead.

Finally, let's explore the last empty value,

which you might be already

familiar with, the empty string.

This is a string without any characters inside

of it and it can be built in a few ways,

such as with single quotes or

double quotes with no characters in between them.

In this video, you learned about

the differences between null, undefined,

and empty strings, as well as some

of the common situations in which they appear.

**Exercise: Error prevention**

**Instructions**

**Task 1: Code a function declaration**

You need to code a function declaration named *addTwoNums*, which accepts numbers *a* and *b* and console logs *a + b*.

**Task 2: Invoke the *addTwoNums* function with a number and a string**

You need to invoke the *addTwoNums* using the following arguments: *5* and *"5"*.

**Task 3: Update the *addTwoNums* function with a try...catch block**

Add the try and catch blocks inside the function definition's body. For now, just make sure that the console log of *a + b* is inside the try block. Additionally, the *catch* block should catch an error named *err* and, inside the body of the *catch* block, you need to console log the *err* value.

**Task 4: If the passed-in arguments are not numbers, throw an error**

If either of the arguments passed to the *addTwoNums* are not numbers, you'll throw an error.

Specifically, code a conditional with the following logic:

* if the *typeof* the *a* parameter is not equal to *'number'*, throw a new *ReferenceError*. Inside the *ReferenceError*, pass a custom error message of *'the first argument is not a number'*.
* else if the *typeof* the *b* parameter is not equal to *'number'*, throw a new *ReferenceError*. Inside the *ReferenceError*, pass a custom error message of *'the second argument is not a number'*.
* else, console log *a + b*

Once you've completed this task, all the code inside the *try* block will be inside these conditional statements.

**Task 5: Update the catch block**

Inside the catch block, update the code from *console.log(err)* to *console.log("Error!", err)*.

**Task 6: Invoke the *addTwoNums* function**

Invoke the *addTwoNums* function using *5* and *"5"* as arguments.

**Task 7: Add another console log under the *addTwoNums* function invocation**

Add another line of code that console logs the string *"It still works"*.

/\*

function addTwoNums(a, b)

{

    console.log(a + b);

}

addTwoNums(5, "5")

\*/

function addTwoNums(a, b)

{

    try

    {

        if(typeof(a) != 'number')

        {

            throw new ReferenceError('the first argument is not a number')

        }

        else if(typeof(b) != 'number')

        {

            Throw new ReferenceError('the second argument is not a number')

        }

        else

        {

            console.log(a + b);

        }

    }

    catch(err)

    {

        console.log("Error " + err);

    }

}

addTwoNums(5, "5")

console.log("It still works")

**Exercise: Defensive programming**

Defensive programming is all about *assuming* that all the arguments a function will receive are of the wrong type, the wrong value or both.

In other words, you are assuming that things will go wrong and you are proactive in thinking about such scenarios before they happen, so as to make your function less likely to cause errors because of faulty inputs.

How would you then refactor the function given below with defensive programming in mind?

For this exercise, let's make sure that both of the arguments that are passed in satisfy the following criteria:

* The length of the *word* parameter cannot be less than *2*.
* The length of the *match* parameter must be *1*.
* The type of both the *word* and the *match* parameters must be *string*.

Here are the tasks to complete:

1. Just above the *for* loop in the *letterFinder* function definition, declare a variable named *condition1* and assign to it the following code: *typeof(word) == 'string' && word.length >= 2*.
2. Declare a variable named *condition2* on the next line and assign to it and assign to it a check that makes sure that the type of *match* is a string AND that the length of the *match* variable is equal to *1*.
3. Write an if statement on the next line that checks that *condition1* is *true*, and *condition2* is *true*
4. Move the rest of the function's body into the if statement you wrote in the previous step.
5. Code an "else" block after the "if" condition and console.log the following: *"Please pass correct arguments to the function."*.
6. As a failing test, run the *letterFinder* function and pass it with any two numbers as arguments.
7. As a passing test, run the *letterFinder* funciton and pass it with correct arguments, such as: *letterFinder("cat", "c")*.

function letterFinder(word, match) {

    var condition1 = typeof(word) == 'string' && word.length >= 2;

    var condition2 = typeof(match) == 'string' && match.length == 1;

    if(condition1 && condition2) {

        for(var i = 0; i < word.length; i++) {

            if(word[i] == match) {

                console.log('Found the', match, 'at', i)

            } else {

                console.log('---No match found at', i)

            }

        }

    } else {

        console.log("Please pass correct arguments to the function")

    }

}

letterFinder(5,2)

letterFinder("chicken","c")

**Additional resources**

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function)

[MDN try...catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch)

[Iteration protocols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols)

[The Math object on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

[The String object on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

[MDN JavaScript error reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors)

[null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)

[undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)

**Introduction to functional programming**

When you listen to a lecture in class,

the language used is probably very formal.

Grammar rules are observed,

and syntax and vocabulary can

get quite complex based on the topic.

Some other examples of the use of

formal language can be found in

courts and other legal institutions.

Now let's consider the way people

talk in their everyday life.

There's lots of slang,

grammar rules are bent and broken,

and the entire delivery is colloquial.

There are many styles in every human language,

and they all perform

the exact same function, communication.

The same can also be said about computer languages.

They too have various styles,

also known as programming paradigms.

Just like in human languages,

no one style is better suited than the other.

In programming, there are two commonly used paradigms.

Functional programming, sometimes abbreviated as FP,

and object-oriented programming,

sometimes abbreviated as OOP.

You can think of these paradigms as

different approaches to writing code.

But the result is still the same,

instructing a computer to perform a set of operations.

In this lesson, you will

learn about functional programming.

Let's now focus on the functional programming style.

There is a clear distinction

between data and functions in

functional programming as data

can exist outside of functions.

For example, you may recall that when functions need

some data you pass them

the values in the form of arguments.

Then the function perform some work

on the given data and returns

some values that you can then

use somewhere else in your program.

An alternative paradigm is

the object-oriented programming paradigm,

where you combine both data and functions into objects.

You'll learn more about

object oriented programming later.

Now let's examine a practical implementation of

the functional programming style using

a coding example to calculate currency conversion.

In this example of functional programming,

let me start by giving my program some data.

The purpose of the program is

to perform currency conversions.

Notice that I already have a variable declared with

the name currencyOne and assigned it the value of 100.

Next, I'll add two more variables, currencyTwo,

which is assigned a value of 0,

and exchangeRate, which is set to 1.2.

Notice that I have declared

all these variable names using CamelCase.

This is where the first letter of

the first word is lowercase and

the first letter of the subsequent words are

without spaces and in uppercase.

Next, I can create the function that will

work with these variables and operate on this data.

First, I create a function named

convertCurrency that accepts two parameters,

amount and rate.

Then in the function's body,

I returned the result of

the amount multiplied by the rate.

Basically, I'm coding it to multiply

the two values it receives and return the result.

Next, I need to update the value of currencyTwo,

which was previously defined on line four,

assigned to the value of 0.

To do this, I assign the value of

currencyTwo to the result of the function I just created.

Recall that when the function is invoked,

it will return a value which will be

assigned to the variable currencyTwo.

It's important to remember that I need to also

pass the two arguments required for my function to work.

In this example, I'm sending

the variables currencyOne and exchangeRate.

Basically, I'm passing the values 100 and

1.2 to the function so it

can perform its calculation and return the result.

Finally, I console log the value

of currencyTwo to test my code.

Success, 120 is output to the console,

the result of 100 multiplied by 1.2.

In this lesson, you learned about

functional programming and how it can be used

to solve a problem by separating

data from functions. Great job.

**Return values from functions**

Many functions, by default, return the value of *undefined*.

An example is the *console.log()* function.

If I run:

console.log('Hello');

... here's the output in the console:

Hello

undefined

Because the *console.log()* function is built so as to not have the explicitly set return value, it gets the default return value of *undefined*.

I'll now code my own implementation of *console.log()*, which doesn't return the value of *undefined*:

function consoleLog(val) {

    console.log(val)

    return val

}

I'm using the *console.log()* function inside my custom *consoleLog* function declaration. And I'm specifying it to return the value of its argument.

Now when I run my custom *consoleLog()* function:

consoleLog('Hello')

I get the following output:

Hello

'Hello'

So, the value is output in the console, but it's also returned.

Why is this useful?

It's useful because I can use return values from one function inside another function.

Here's an example.

I'll first code a function that returns a double of a number that it received:

function doubleIt(num) {

    return num \* 2

}

Now I'll code another function that builds an object with a specific value:

function objectMaker(val) {

    return {

        prop: val

    }

}

I can call the *objectMaker()* function with any value I like, such as:

objectMaker(20);

The returned value will be an object with a single *prop* key set to *20*:

{prop:20}

Now consider this code:

doubleIt(10).toString()

The above code returns the number *20* as a string, that is: *"20"*.

I can even combine my custom function calls as follows:

objectMaker( doubleIt(100) );

This will now return the following value:

{prop: 200}

What does all of this mean?

It means that by JavaScript allowing me to use the *return* keyword as described above, I can have multiple function calls, returning data and manipulating values, based on whatever coding challenge I have in front of me.

Being able to return custom values is one of the foundations that makes functional programming possible.

**Function calling and recursion**

Treadmills are a useful piece of equipment for doing cardio at the gym, or

when running outdoor isn't an option.

They are also easily started with the push of a button.

Imagine however, if there was no way to control when a treadmill stopped,

that wouldn't be ideal, would it?

Fortunately, most treadmills have a function that tells the treadmill to stop

after a specified amount of time.

Thank goodness.

In JavaScript, functions that repeat tasks are similarly helpful,

unless they run endlessly.

In this video, I'll demonstrate what recursive functions are, and help you

understand how to write them properly to avoid getting stuck in an infinite loop.

To call, invoke, or execute a function means instructing it to follow

the code inside of it one line at a time.

To demonstrate, let's first build an example.

Type function example, parentheses, then left curly brace.

Press enter to advance to the next line and type console.log.

Inside of the parentheses, type quote line one, unquote, and

then close off with a semi colon.

I now have one line of code, but let's add a few more.

On separate lines, I am put two more console.log lines as I did before,

but this time containing the strings line two and line three respectively.

I close the function by inputting a right curly brace.

I've built a function called example with three lines.

When I run it, each of the lines will be executed one at a time in sequence

producing three strings.

To make things interesting, let's add one more line to the example function.

This time, type the name of the function itself.

So that's example, parentheses, semicolon.

Now, if I run the function again, it will repeat in an infinite loop.

Obviously this wouldn't be useful.

So let's improve it, and ensure that it won't run endlessly.

I add a new line before the function and let counter equal sign three semi colon.

Next, I delete my three console.log lines,

which I'll replace with some different code.

For the first line, I type console.log, and

then counter inside of the parentheses.

On the second line, I type counter equal sign, counter minus 1 semi colon.

And finally, for the third line, I start by typing if, and

then inside of the parentheses, I type counter triple equal sign, zero.

And then I type return in a semi colon.

If I call the function this time, it will log the numbers 3, 2, 1, and then stop.

When a function calls itself, this is what's known as recursion.

Recursion is an alternative way to run repetitive code without the use of loops.

Next time you see someone running on a treadmill,

imagine a function running in the background.

It might be calling itself to continue running until its condition is meant

to stop.

In this video, you learned about the uses and potential problems of recursive

functions, and how to write them so that they don't run endlessly.

**Introduction to scope**

Scope is all about code accessibility.

It determines which parts of the code are

accessible and which parts are inaccessible.

For example, what variables

can a function access within code?

Over the next few minutes,

you will explore the concept of scope and learn

about how the scope chain works within JavaScript.

You will also explore some of the different scope types,

such as global and local.

A nice way to think about how scope works in

JavaScript is a two-way mirror glass.

This is a piece of glass where

only one side is transparent.

For example, if a restaurant uses two-way glass,

people outside the restaurant can't see what's happening

inside but the people

inside can see what is happening outside.

The process is similar to how scope works in JavaScript.

For example, the code that exists

outside of a function is referred to as global scope,

and all the code inside of a function is

known as local scope or function scope.

If a variable is defined within a function,

then you can say it's scoped to that function.

This is also known as local scope.

For example, I could define a variable

named localvar and place it within a function.

I could then say that this variable is

scoped to the function in which it was created.

Each function keeps a reference to its parent scope.

This chain of scope references is

referred to as the scope chain.

You're now familiar with scope and

how the scope chain works in JavaScript.

You can also identify how

some of the different types of scope work,

including global and local,

also known as function scope.

**The functional programming paradigm**

**Learning Objectives**

* Be able to explain that there are several programming paradigms
* Be able to explain the basic difference between the two predominant programming paradigms: functional programming and object-oriented programming
* Understand, at a very high level, how the functional programming paradigm works

"There are actually several styles of coding, also known as **paradigms**. A common style is called **functional programming**, or FP for short.

In functional programming, we use a lot of functions and variables.

function getTotal(a,b) {

    return a + b

}

var num1 = 2;

var num2 = 3;

var total = getTotal(num1, num2);

When writing FP code, we keep data and functionality separate and pass data into functions only when we want something computed.

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

In functional programming, functions return new values and then use those values somewhere else in the code.

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

console.log(distance); // <====== THIS HERE!

Another style is **object-oriented programming (OOP)**. In this style, we group data and functionality as properties and methods inside objects.

For example, if I have a *virtualPet* object, I can give it a *sleepy* property and a *nap()* method:

var virtualPet = {

    sleepy: true,

    nap: function() {}

}

In OOP, methods **update properties** stored in the object instead of generating new return values.

For example, if I check the *sleepy* property on the *virtualPet* object, I can confirm that it's set to *true*.

However, once I've ran the *nap()* method on the *virtualPet* object, will the *sleepy* property's value change?

var virtualPet = {

    sleepy: true,

    nap: function() {

        this.sleepy = false

    }

}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false

* OOP helps us model real-life objects. It works best when the grouping of properties and data in an object makes logical sense - meaning, the properties and methods "belong together".
* Note that the goal here is not to discuss OOP in depth; instead, I just want to show you the simplest explanation of what it is and how it works, in order to make the single most important distinction between FP and OOP.
* To summarize this point, we can say that the Functional Programming paradigm works by keeping the data and functionality separate. It's counterpart, OOP, works by keeping the data and functionality grouped in meaningful objects.

There are many more concepts and ideas in functional programming.

Here are some of the most important ones:

* First-class functions
* Higher-order function
* Pure functions and side-effects

There are many other concepts and principles in functional programming, but for now, let's stick to these three.

**First-class functions**

It is often said that functions in JavaScript are “first-class citizens”. What does that mean?

It means that a function in JavaScript is just another value that we can:

* pass to other functions
* save in a variable
* return from other functions

In other words, a function in JavaScript is just a value - from this vantage point, almost no different then a string or a number. For example, in JavaScript, it's perfectly normal to pass a function invocation to another function. To explain how this works, consider the following program.

*function addTwoNums(a, b) {*

*console.log(a + b)*

*}*

*function randomNum() {*

*return Math.floor((Math.random() \* 10) + 1);*

*}*

*function specificNum() { return 42 };*

*var useRandom = true;*

*var getNumber;*

*if(useRandom) {*

*getNumber = randomNum*

*}*

*else {*

*getNumber = specificNum*

*}*

*addTwoNums(getNumber(), getNumber())*

* I start the program with the *addTwoNums()* function whose definition I've already used earlier in various variations. The reason why this function is a recurring example is because it's so simple that it helps explain concepts that otherwise might be a bit harder to grasp.
* Next, I code a function named *randomNum()* which returns a random number between 0 and 10. I then code another function named *specificNum()* which returns a specific number, the number 42. Next, I save a variable named *useRandom*, and I set it to the boolean value of *true*. I declare another variable, named *getNumber*.
* This is where things get interesting.
* On the next several lines, I have an if else statement. The if condition is executed when the value of *useRandom* is set to *true*. If that's the case, the entire *randomNum()* function's declaration is saved into the *getNumber* variable. Otherwise, I'm saving the entire *specificNum()* function's declaration into the *getNumber* variable.
* In other words, based on the *useRandom* being set to *true* or *false*, the *getNumber* variable will be assigned either the *randomNum()* function declaration or the *specificNum()* function declaration.
* With all this code set, I can then invoke the *addTwoNums()* function, passing it the invocation of the *getNumber()* variables as its first and second arguments.
* **This works because functions in JavaScript are truly first-class citizens, which can be assigned to variable names and passed around just like I would pass around a string, a number, an object, etc.** Note: most of the code inside the *randomNum()* function declaration comes from a previous lesson, namely the lesson that discussed the Math object in JavaScript. This brings me to the second foundational concept of functional programming, which is the concept of higher-order functions.

**Higher-order functions**

A higher-order function is a function that has either one or both of the following characteristics:

* It accepts other functions as arguments
* It returns functions when invoked
* There's no "special way" of defining higher-order functions in JavaScript. It is simply a feature of the language. The language itself allows me to pass a function to another function, or to return a function from another function.
* Continuing from the previous section, consider the following code, in which I'm re-defining the *addTwoNums()* function so that it is a higher-order function:

*function addTwoNums(getNumber1, getNumber2) {*

*console.log(getNumber1() + getNumber2());*

*}*

* You can think of the above function declaration of *addTwoNums* as describing how it will deal with the *getNumber1* and *getNumber2* inputs: once it receives them as arguments, it will then attempt invoking them and concatenating the values returned from those invocations.
* For example:

*addTwoNums(specificNum, specificNum); // returned number is 84*

*addTwoNums(specificNum, randomNum); // returned number is 42 + some random number*

**Pure functions and side-effects**

Another concept of functional programming are pure functions.

A pure function returns the exact same result as long as it's given the same values.

An example of a pure function is the *addTwoNums()* function from the previous section:

function addTwoNums(a, b) {

    console.log(a + b)

}

This function will always return the same output, based on the input. For example, as long as we give it a specific value, say, a *5*, and a *6*:

addTwoNums(5,6); // 11

... the output will always be the same.

Another rule for a function to be considered pure is that it should not have side-effects. A side-effect is any instance where a function makes a change outside of itself.

This includes:

* changing variable values outside of the function itself, or even relying on outside variables
* calling a Browser API (even the console itself!)
* calling *Math.random()* - since the value cannot be reliably repeated

The topic of pure and impure functions can get somewhat complex.

For now, it's sufficient to know that this concept exists and that it is related to functional programming.

**Lab: Building a Functional Program**

// Task 1: Build a function-based console log message generator

function consoleStyler(color, background, fontSize, txt) {

    var message = "%c" + txt;

    var style = `color: ${color};`;

    style += `background: ${background};`;

    style += `font-size: ${fontSize};`

    console.log(message, style)

}

// Task 2: Build another console log message generator

function celebrateStyler(reason) {

    var fontStyle = "color: tomato; font-size: 50px";

    if (reason == "birthday"){

        console.log(`%cHappy birthday`, fontStyle);

    } else if (reason == "champions") {

        console.log(`%cCongrats on the title!`, fontStyle);

    } else {

        console.log(message, style);

    }

}

// Task 3: Run both the consoleStyler and the celebrateStyler functions

consoleStyler('#1d5c63', '#ede6db', '40px', 'Congrats!');

celebrateStyler('birthday')

// Task 4: Insert a congratulatory and custom message

function styleAndCelebrate(color, background, fontSize, txt, reason) {

    consoleStyler(color, background, fontSize, txt);

    celebrateStyler(reason);

}

// Call styleAndCelebrate

styleAndCelebrate('ef7c8e', 'fae8e0', '30px', 'You made it!', 'champions')

**Scoping with var, let and const**

You might recall that scope

relates to code accessibility.

It determines which part of the code are

accessible by different parts of the program.

In this video you will explore scoping in more detail,

and learn about two more scope types and how they

work with the keywords var, let, and const.

You may recall that all the code outside of

functions is referred to as global scope,

and all the code inside of

a function is known as local scope.

Local scope states that a variable is only

accessible in the function where it is declared.

In the ES5 version of JavaScript,

only functions can build local scope.

The ES6 version of JavaScript introduced

a new variety of scope known as the block scope.

Block scope states that a variable declared in a block of

code is only accessible inside that block.

All the other code outside of

the code block cannot access it.

Block scope is built when you declare

variables using let or const.

In other words when you build

variables with let or const,

they become immediately scoped to

the code block they were created in.

The scope of these variables

is contained within curly braces.

For example, you can declare

two separate variables with the same name.

If one is declared inside

curly braces and the other is not,

you can then only access

these variables inside their scope.

Before ES6, the only way to declare

a variable in JavaScript was to use the var keyword.

The var keyword is very lenient.

Let's outline some characteristics

of variables that are declared with var.

First, you can use it in

your code even before it is declared.

Also, you can redeclare

the same variable when you use VAR.

The variables are scoped to a function R if

they are declared outside

the function their scope is global.

With ES6, the suggested way to declare

variables is to use the let or const keywords.

Its syntax is very similar to the var syntax.

Only the keyword is replaced.

For example, let's say you want to declare

a variable named user and assign

it to a string set to the value of Miranda.

In ES5, you use the keyword var,

the variable user, the equal operator,

and then the string value Miranda inside double quotes.

In ES6 with let and const,

you use the same syntax,

only changing the var keyword to either let or const.

Notice that the syntax is very similar.

You might be wondering what's the difference between var,

let, and const.

The simplest explanation is that

the behavior of let and const is more strict.

With a let or a const variable,

you cannot use it in your code before you declare it.

You can redeclare it using

the variable keyword like you can with var.

Finally, it's scoped to the block,

even within if statements and loops,

like the far or while loops.

If you are new to JavaScript,

it may be confusing as to when to use either var,

let, or const.

A pro tip is to remember it's like this.

Use let if the value might change in the future,

and use const if the value will never change.

In this video, you learned how the scope of variables

changes when you use the keyword var in ES5 JavaScript,

and when you use the keywords let and const

in ES6 JavaScript. Great job.

**Comparing var, let and const**

In this video, you'll learn about the var, let, and const variables and

the different rules that they are bound to.

Let's start with the var keyword.

I've typed some example code that is currently commented out but

we'll go through it one point at a time.

A variable declared with the var keyword can be accessed before initialization as

long as the variable is eventually initialized somewhere in our code.

To confirm that, let's try to console log a variable that hasn't been declared or

initialized.

I type console.log(user) and click the Run button.

This returns reference error.

User is not defined which is probably expected.

However, suppose I use the var keyword to declare the variable user

as my last line of code and rerun the code.

Notice that undefined is output but no error.

This means that the JavaScript engine continues to run.

Let me clear my window here.

And another thing to know about the var keyword is that we can declare and

redeclare the same var variable without errors.

Let me in comment lines 9-13 in my code to demonstrate.

Notice that each line uses the var keyword to repeatedly declare the user variable

and assigns it to a different string value Mary, Joanna, and finally, Mark.

I'll comment out line 15.

When I run my code, it outputs the first value undefined and

then the value of the latest reassignment, in this case the string Mark.

Let's move on to the let keyword.

A key difference is that you cannot access a let variable before declaring it.

For example, in lines 4 and

5 suppose I run a console log on the user variable as our first line.

In that case, I get the reference error cannot access user before initialization.

What we can do however, is declare an unassigned variable with let.

Running a console log by uncommenting lines 8 and 9 works without problems and

returns undefined.

However, once you declare a variable threat, you cannot redeclare it.

Trying to do so by uncommenting line 12 will produce the syntax error,

identifier user has already been declared.

While you cannot redeclare a let variable, you can reassign it.

For example, notice lines 8 and 15,

initially it is set to undefined and then to the string value of Anna.

Now if I run my code,

notice that undefined is output first followed by the string Anna.

Finally, let me switch tabs again to demonstrate

the const keyword which is the strictest.

A const variable must be initialized.

For example, on lines 4 and 5,

you get the syntax error missing initializer in const declaration.

Next, let's comment out lines 4 and 5 and uncomment lines 8 and 9.

As you can see, you can access a const variable before initialization or

else you'll get a reference here.

Next uncommon line 9 as well as line 12.

A const variable can't be re declared as it throws the type error assignment to

constant variable.

Okay, and that's it for

this video comparing variable declaration using var, let and const.

It might seem confusing at the beginning to know which type to choose.

Try to remember it like this.

The Var keyword is the most lenient, while the Const keyword is the strictest.

The best choice and use, usually depends on whether or

not you will be reassigning values.

In modern JavaScript, I'd advise you to pick either let or

const based on whether they will be reassigned or not.

Why not give it a try in your own practice code?

**Additional resources**

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN Functions Guide](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)

[MDN Glossary: Recursion](https://developer.mozilla.org/en-US/docs/Glossary/Recursion)

[MDN Glossary: Scope](https://developer.mozilla.org/en-US/docs/Glossary/Scope)

[Functional Programming in JavaScript](https://www.toptal.com/javascript/functional-programming-javascript)

[MDN: First-class functions](https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function)

**Introduction to object-oriented programming**

In programming, there is something known as the programming paradigms.

You can think of this as a classification, a style or

just a general way to write code.

You may already be familiar with one of these paradigms.

Functional programming, in this video, you will learn about another popular one.

The object oriented programming paradigm, often referred to as OOP.

OOP revolves around the idea of organizing our programs using objects

to group related data and functionality.

This is in contrast to the functional programming approach, where the data used

in the app needs to be kept separate from functions that operate on that data.

Let's explore this concept further using some code examples from the two paradigms,

suppose we are asked to write some code that calculates the total cost of

buying a pair of shoes.

The code needs to calculate the total price which is the shoes multiplied

by the tax amount.

To code this solution, you decide that you need variables to store the values for

shoes and tax and total price.

You need a function which you will call total price to perform

the calculation of multiplying the shoes by the tax.

Finally, you need to be able to output the result.

Using the functional programming approach,

you clearly separate a program's data from functions that work on that data.

With the OOP approach, you create an object and store all data related to

that object including variables, functions and output statements.

For example, you create an object named purchase one.

You may recall that functions inside objects are known as methods.

Now that the purchase 1 object is created.

You access the total price method on the purchase1 object using the dot notation.

Then, you invoke the total price method which works with data inside

the purchase 1 object and returns the result of 120.

In fact, you can access any data that the purchase 1 object has.

For example, using the dot notation,

you can access the shoes price data and the state tax data.

An advantage to using the OoP approach is that you can build as many

objects as you need.

For example, you can build another purchase object and name it purchase2.

Once created, you can also access the total price method on the purchase2

object just like you did on the purchase 1 object previously.

You may notice, that the total price method was almost the same in both

the purchase 1 and the purchase 2 objects.

This means that you can improve the objects so

that both methods are identical.

And you can do that by using this keyword which you may recall.

It essentially means this object.

It's important to note that the code still works exactly the same as before.

Okay, so you may be wondering why did I go through the trouble of updating

the code of purchase 1 dot choose price to this dot choose price.

Well, the advantage here is that rather than having to think about

the name of the object whose shoes priced property I'm trying to access,

I just used the alias of the current objects name, namely, this keyword.

And now I can just copy the total price method from the purchase 1 object and

reuse it in the purchase 2 object.

So, using that this keyword allows me to not really care about the current

objects name, which is an improvement to the previous code I had.

However, programmers are always eager to avoid wasting resources when writing code.

And coding a custom method on each of the objects I built is wasteful.

The solution to this is to code my objects using some sort of a template.

You will learn how to create this later in this lesson using something called

a class.

For now, let's explore object oriented programming principles with

the example of calculating the total cost of buying a pair of shoes.

Before starting, let's first quickly revisit another programming style.

Functional programming,

this should help you understand the differences between the two paradigms.

I start with var shoes equals 100 and

I also add var state tax equals 1.2.

Next, I declare a function,

which returns the value of the shoes multiplied by the value of the tax.

This is written as function total price, with shoes and tax in parenthesis.

Inside of curly braces, I add return shoes, asterix tax.

Now, I'm declaring the variable to pay which invokes the total price function and

passes the shoes variable value as well as the state tax variable value.

Finally, I cancel log the to pay variable.

When I run the code, the output is 120.

So that's an example of the functional paradigm.

But this time let's build something similar with the OOP paradigm.

I go to a separate file where I declare the variable, purchase 1 and

store an object literal inside of it with curly braces.

I didn't add the property shoes and set it to 100,

followed by state tax, which is set to 1.2.

The last property is total price,

which is set to a function to declare a calculation variable.

This variable is equal to the purchase1 dot shoes property

multiplied by the purchase 1 dot state tax property.

The 2nd part of the function console logs the string total

price along with the value of the calculation variable.

Online 10, I call the function with purchase 1 dot total price parenthesis.

Which I expect to return a value of 120.

I run the code and that's indeed what I get.

On another file, I'll build another OOP example, this one starts with

the variable purchase 2, which has the same structure as purchase 1.

However, that shoes value is changed to 50.

In the total price function, the calculation variable is updated to

access the shoes and state tax properties from purchase 2.

When I invoked the total price method on the purchase 2 object and

run the code, It returns a value of 60.

Between these two OOP examples, you may have noticed that aside from

accessing the shoes and state tax properties in different objects,.

The two methods are completely the same.

This means that I can improve my objects to make both methods identical while

getting the same results.

I can do this using the this keyword which essentially means this object.

Let's move to another file, to examine how that works.

In this file, I had the same purchase 1 and

purchased 2 objects from before with all of the same values.

in the total price function of purchase 1,

I replaced the references to purchase 1 with the this keyword.

I can then copy the total price method from purchase1 and

paste it into the purchase 2 object, since they are now identical.

I run this code and it gives me a total price of 120 for

purchase 1 and 60 for purchase 2.

This is an improvement,

over my original code from the perspective of reducing waste.

Instead of writing custom methods for every object,

this approach allows for the reuse of existing code.

However, although I'm reusing the existing code,

I'm now repeating the same method on each new object that is built.

That is wasteful and programs need to be efficient.

This is where making templates for objects comes in.

In JavaScript, one of the most elegant ways

to efficiently build new objects is by using classes.

Congratulations.

In this video, you learned about object orientated programming and

how it differs from functional programming.

With the object orientated approach,

you can code more efficiently by reusing existing code.

**Classes**

In programming there are situations where you need to build many objects that

have a certain specific set of properties and methods.

For example, you might need to build hundreds of car objects for

a car racing game.

To code this efficiently, you can use something called classes.

They are essentially a blueprint that you can repeatedly use to build new objects of

a certain kind, as many times as you like.

In this video, you will learn about classes.

In java script any class is built using the class keyword, followed by the name

of the class starting with a capital letter and a pair of curly braces.

Inside of the curly braces you have the constructor function which

accepts as many parameters as needed.

The role of the constructor function is to assign the passed in parameters

to the future objects properties.

It is the constructor function that is used when instantiating new objects,

instances of a given class.

After the constructor is defined, you add as many methods as you want.

It's important to remember that you don't use the function keyword here.

Just the name of the method is needed.

Once the class definition is ready, you can start building the car object,

now that you have instantiation the car class and

saved the instant of the class as car one.

You have access to its methods and properties.

For example, to access the turbo on method,

you type the name of the object which is car one, then the dot and

then the name of the method that exists on that object.

In this example, the turbo on method, with a pair of parenthesis to invoke it.

When invoked, the turbo on method will work with the available data in

the car one object to produce its output.

Like with regular functions, you can also pass parameters to the class methods and

then use them the same as with regular functions.

In this video, you learned about classes and

how they can be used to build multiple object instances with specific properties.

**Object Oriented Programming principles**

In this reading, you'll learn about the benefits of object-oriented programming (OOP) and the OOP principles.

**The Benefits of OOP**

There are many benefits to using the object-oriented programming (OOP) paradigm.

OOP helps developers to mimic the relationship between objects in the real world. In a way, it helps you to reason about relationships between things in your software, just like you would in the real world. Thus, OOP is an effective approach to come up with solutions in the code you write. OOP also:

* Allows you to write modular code,
* Makes your code more flexible and
* Makes your code reusable.

**The Principles of OOP**

The four fundamental OOP principles are inheritance, encapsulation, abstraction and polymorphism. You'll learn about each of these principles in turn. The thing to remember about Objects is that they exist in a hierarchal structure. Meaning that the original base or super class for everything is the Object class, all objects derive from this class. This allows us to utilize the Object.create() method. to create or instansiate objects of our classes.

class Animal { /\* ...class code here... \*/ }

var myDog = Object.create(Animal)

console.log (Animal)

A more common method of creating obbjects from classes is to use the *new* keyword. When using a default or empty constructor method, JavaScript implicitly calls the Object superclass to create the instance.

class Animal { /\* ...class code here... \*/ }

var myDog = new Animal()

console.log (Animal)

This concept is explored within the next section on inheritance

**OOP Principles: Inheritance**

Inheritance is one of the foundations of object-oriented programming.

In essence, it's a very simple concept. It works like this:

1. There is a base class of a "thing".
2. There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class")
3. There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.

Note that each sub-class inherits from its super-class. In turn, a sub-class might also be a super-class, if there are classes inheriting from that sub-class.

All of this might sound a bit "computer-sciency", so here's a more practical example:

1. There is a base class of "Animal".
2. There is another class, a sub-class inheriting from "Animal", and the name of this class is "Bird".
3. Next, there is another class, inheriting from "Bird", and this class is "Eagle".

Thus, in the above example, I'm modelling objects from the real world by constructing relationships between Animal, Bird, and Eagle. Each of them are separate classes, meaning, each of them are separate blueprints for specific object instances that can be constructed as needed.

To setup the inheritance relation between classes in JavaScript, I can use the *extends* keyword, as in *class B extends A*.

Here's an example of an inheritance hierarchy in JavaScript:

class Animal { /\* ...class code here... \*/ }

class Bird extends Animal { /\* ...class code here... \*/ }

class Eagle extends Bird { /\* ...class code here... \*/ }

**OOP Principles: Encapsulation**

In the simplest terms, encapsulation has to do with making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.

For example, when I run the following code:

"abc".toUpperCase();

I don't really need to worry or even waste time thinking about how the *toUpperCase()* method works. All I want is to use it, since I know it's available to me. Even if the underlying syntax - that is, the implementation of the *toUpperCase()* method changes - as long as it doesn't break my code, I don't have to worry about what it does in the background, or even how it does it.

**OOP Principles: Abstraction**

Abstraction is all about writing code in a way that will make it more generalized.

The concepts of encapsulation and abstraction are often misunderstood because their differences can feel blurry.

It helps to think of it in the following terms:

* An abstraction is about extracting the *concept* of what you're trying to do, rather than dealing with a specific manifestation of that concept.
* Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.

While both the encapsulation and abstraction are important concepts in OOP, it requires more experience with programming in general to really delve into this topic.

For now, it's enough to be aware of their existence in OOP.

**OOP Principles: Polymorphism**

Polymorphism is a word derived from the Greek language meaning "multiple forms". An alternative translation might be: "something that can take on many shapes".

So, to understand what polymorphism is about, let's consider some real-life objects.

* A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
* Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on a door.

The above concepts can be coded in JavaScript as follows:

const bicycle = {

    bell: function() {

        return "Ring, ring! Watch out, please!"

    }

}

const door = {

    bell: function() {

        return "Ring, ring! Come here, please!"

    }

}

So, I can access the *bell()* method on the *bicycle* object, using the following syntax:

bicycle.bell(); // "Get away, please"

I can also access the *bell()* method on the *door* object, using this syntax:

door.bell(); // "Come here, please"

At this point, one can conclude that the exact same name of the method can have the exact opposite intent, based on what object it is used for.

Now, to make this code truly polymorphic, I will add another function declaration:

function ringTheBell(thing) {

    console.log(thing.bell())

}

Now I have declared a *ringTheBell()* function. It accepts a *thing* parameter - which I expect to be an object, namely, either the *bicycle* object or the *door* object.

So now, if I call the *ringTheBell()* function and pass it the *bicycle* as its single argument, here's the output:

ringTheBell(bicycle); // Ring, ring! Watch out, please!

However, if I invoke the *ringTheBell()* function and pass it the *door* object, I'll get the following output:

ringTheBell(door); // "Ring, ring! Come here, please!"

You've now seen an example of the exact same function producing different results, **based on the context** in which it is used.

Here's another example,the concatenation operator, used by calling the built-in *concat()* method.

If I use the *concat()* method on two strings, it behaves exactly the same as if I used the *+* operator.

"abc".concat("def"); // 'abcdef'

I can also use the *concat()* method on two arrays. Here's the result:

["abc"].concat(["def"]); // ['abc', 'def']

Consider using the *+* operator on two arrays with one member each:

["abc"] + ["def"]; // ["abcdef"]

This means that the *concat()* method is exhibiting polymorphic behavior since it behaves differently based on the context - in this case, based on what data types I give it.

To reiterate, polymorphism is useful because it allows developers to build objects that can have the exact same functionality, namely, functions with the exact same name, which behave exactly the same. However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

Here's an example of polymorphism using classes in JavaScript:

class Bird {

    useWings() {

        console.log("Flying!")

    }

}

class Eagle extends Bird {

    useWings() {

        super.useWings()

        console.log("Barely flapping!")

    }

}

class Penguin extends Bird {

    useWings() {

        console.log("Diving!")

    }

}

var baldEagle = new Eagle();

var kingPenguin = new Penguin();

baldEagle.useWings(); // "Flying! Barely flapping!"

kingPenguin.useWings(); // "Diving!"

The *Penguin* and *Eagle* sub-classes both inherit from the *Bird* super-class. The *Eagle* sub-class inherits the *useWings()* method from the *Bird* class, but extends it with an additional console log. The *Penguin* sub-class doesn't inherit the *useWings()* class - instead, it has its own implementation, although the *Penguin* class itself does extend the *Bird* class.

Do some practice with the above code, try creating some of your own classes. (hint : think about things you know from everyday life)

**Constructors**

* JavaScript has a number of built-in object types, such as:
  + *Math*, *Date*, *Object*, *Function*, *Boolean*, *Symbol*, *Array*, *Map*, *Set*, *Promise*, *JSON*, etc.
* These objects are sometimes referred to as "native objects".
* Constructor functions, commonly referred to as just "constructors", are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized.
* To use a constructor function, I must prepend it with the operator *new*.
* For example, to create a new instance of the *Date* object, I can run: *new Date()*. What I get back is the current datetime, such as:
* *Thu Feb 03 2022 11:24:08 GMT+0100 (Central European Standard Time)*
* However, not all the built-in objects come with a constructor function. An example of such an object type is the built-in *Math* object.
* Running *new Math()* throws an *Uncaught TypeError*, informing us that *Math is not a constructor*.
* Thus, I can conclude that some built-in objects do have constructors, when they serve a particular purpose: to allow us to instantiate a specific instance of a given object's constructor. The built-in *Date* object is perfectly suited for having a constructor because each new date object instance I build should have unique data by definition, since it's going to be a different timestamp - it's going to be built at a different moment in time.
* Other built-in objects that don't have constructors, such as the *Math* object, don't need a constructor. They're just static objects whose properties and methods can be accessed directly, from the built-in object itself. In other words, there is no point in building an instance of the built-in *Math* object to be able to use its functionality.
* For example, if I want to use the *pow* method of the *Math* object to calculate exponential values, there's no need to build an instance of the *Math* object to do so. For example, to get the number 2 to the power of 5, I'd run:
* *Math.pow(2,5); // --> 32*
* There's no need to build an instance of the *Math* object since there would be nothing that needs to be stored in that specific object's instance.
* Besides constructor functions for the built-in objects, I can also define custom constructor functions.

Here's an example:

function Icecream(flavor) {

    this.flavor = flavor;

    this.meltIt = function() {

        console.log(`The ${this.flavor} icecream has melted`);

    }

}

Now I can make as many icecreams as I want:

function Icecream(flavor) {

    this.flavor = flavor;

    this.meltIt = function() {

        console.log(`The ${this.flavor} icecream has melted`);

    }

}

let kiwiIcecream = new Icecream("kiwi");

let appleIcecream = new Icecream("apple");

kiwiIcecream; // --> Icecream {flavor: 'kiwi', meltIt: ƒ}

appleIcecream; // --> Icecream {flavor: 'apple', meltIt: ƒ}

* I've just built two instance objects of *Icecream* type.
* The most common use case of *new* is to use it with one of the built-in object types.
* Note that using constructor functions on all built-in objects is sometimes not the best approach.
* This is especially true for object constructors of primitive types, namely: *String*, *Number*, and *Boolean*.

For example, using the built-in *String* constructor, I can build new strings:

let apple = new String("apple");

apple; // --> String {'apple'}

The *apple* variable is an object of type *String*.

Let's see how the *apple* object differs from the following *pear* variable:

let pear = "pear";

pear; // --> "pear"

* The *pear* variable is a string literal, that is, a primitive Javascript value.
* The *pear* variable, being a primitive value, will always be more performant than the *apple* variable, which is an object.
* Besides being more performant, due to the fact that each object in JavaScript is unique, you can't compare a String object with another String object, even when their values are identical.
* In other words, if you compare *new String('plum') === new String('plum')*, you'll get back *false*, while *"plum" === "plum"* returns true. You're getting the *false* when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where objects are saved.
* Besides not using constructors to build object versions of primitives, you are better off not using constructors when constructing plain, regular objects.
* Instead of *new Object*, you should stick to the object literal syntax: *{}*.
* A RegExp object is another built-in object in JavaScript. It's used to **pattern-match strings** using what's known as "Regular Expressions". Regular Expressions exist in many languages, not just JavaScript.
* In JavaScript, you can built an instance of the RegExp constructor using *new RegExp*.

Alternatively, you can use a pattern literal instead of RegExp. Here's an example of using */d/* as a pattern literal, passed-in as an argument to the *match* method on a string.

"abcd".match(/d/); // null

"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]

Instead of using *Array*, *Function*, and *RegExp* constructors, you should use their array literal, function literal, and pattern literal varieties: *[]*, *() {}*, and */()/*.

However, when building objects of other built-in types, we can use the constructor.

Here are a few examples:

new Date();

new Error();

new Map();

new Promise();

new Set();

new WeakSet();

new WeakMap();

The above list is inconclusive, but it's just there to give you an idea of some constructor functions you can surely use.

Note that there are links provided about RegExp and regular expression in the lesson item titled *"Additional Reading"*.

**Inheritance**

In the real world, inheriting something means acquiring possession,

condition our trade from past generations.

In this video you will learn that inheritance also exists in JavaScript and

the inheritance model revolves around something called the prototype.

You may also be familiar with the concept of a prototype which is often

referred to as an original model from which other forms are developed.

In JavaScript, the prototype is an object that can hold properties to be

shared by multiple other objects.

And this is the basis of how inheritance works in JavaScript.

This is why it's sometimes said that JavaScript implements a prototype of

inheritance model.

Let's explore this further now,

with some code examples to demonstrate inheritance and how to build a prototype.

Consider the following object.

I'm setting the var bird to an object that has three properties.

Each set to the boolean value of true.

The properties are has wings can fly and has feathers.

Using the readily available object create, I can construct new objects.

For example, I've set the eagle1 variable to a call that takes the bird object and

passes it to the object .create method.

I can console log the contents of the eagle1 object now.

Let's run the code and notice an empty object logged to the console.

However, since I used object to create to instantiate the eagle1 object.

I also have access to all the properties of the bird object.

So I canceled log eagle1 has wings and pass eagle1.has wings.

And I also console log eagle1 can fly and pass eagle1.can fly.

And finally eagle1 has feathers passing it eagle1.has feathers.

Let's read from the code once again notice eagle1 has wings true,

eagle1 can fly true and eagle1 has feathers true.

Our output to the console with the object creates syntax.

I can build as many objects as I want and

they will all have the bird objects as their prototype.

Here, notice that I built an eagle2 object and

I've used the bird object as a prototype.

Because I ran the object .create method on it and

I save everything to the eagle2 variable.

It's important to understand that the eagle2 object

also has access to the property stored on the bird object as its prototype.

Let's run this code to confirm and

indeed notice that eagle2 has wings true is output to the console.

I can even add different objects with different behaviors.

For example, I can add a penguin1 object.

I do this by declaring a penguin1 variable and

assigning the result of the object .create method to it.

Penguins are flightless birds so I want to set the can fly property to false.

Thankfully, this is a relatively straightforward process because JavaScript

starts from the object itself when looking for properties to work with.

Then if it can't find it on the object, it looks up to its prototype.

It's important to remember that it doesn't look further if it finds the property on

the immediate object.

This makes for a simple mechanism for overriding inherited properties.

So let's implement this now.

First I set that can fly property on the penguin1 object to false.

And now I can cancel log that penguin1 object.

Notice the output to the console after running the code is that

penguin1 is an object with a can fly property set to false.

Additionally, I can still access all the properties of the prototype.

So accessing that has feathers and can fly properties of penguin1 will

return the values that are stored on the prototype when I run the code.

However, the can fly property is now set on the penguin1 object itself,

so it overrides the can fly property on the prototype.

This override only affects the penguin1 object,

it doesn't change my prototype or other eagle objects.

In this video, you learned about inheritance in JavaScript, although it is

possible to build inheritance using the object create method.

It's probably better to use class syntax for more complex objects and inheritance.

Although under the hood, this syntax still works with prototypes.

It makes sense to use classes as they improve developer experience in

more complex scenarios.

**Creating classes**

* By the end of this reading, you should be able to explain, with examples, the concept of extending classes using basic inheritance to alter behaviors within child classes.
* By now, you should know that inheritance in JavaScript is based around the prototype object.
* All objects that are built from the prototype share the same functionality.
* When you need to code more complex OOP relationships, you can use the *class* keyword and its easy-to-understand and easy-to-reason-about syntax.
* Imagine that you need to code a *Train* class.
* Once you've coded this class, you'll be able to use the keyword *new* to instantiate objects of the *Train* class.
* For now though, you first need to define the *Train* class, using the following syntax:

class Train {}

* So, you use the *class* keyword, then specify the name of your class, with the first letter capitalized, and then you add an opening and a closing curly brace.
* In between the curly braces, the first piece of code that you need to define is the **constructor**:

class Train {

    constructor() {

    }

}

* The *constructor* will be used to build properties on the future object instance of the *Train* class.
* For now, let's say that there are only two properties that each object instance of the *Train* class should have at the time it gets instantiated: *color* and *lightsOn*.

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

}

* Notice the syntax of the constructor. The constructor is a special function in my *Train* class.
* First of all, notice that there is no *function* keyword. Also, notice that the keyword *constructor* is used to define this function. You give your *constructor* function parameters inside an opening and closing parenthesis, just like in regular functions. The names of parameters are *color* and *lightsOn*.
* Next, inside the *constructor* function's body, you assigned the passed-in *color* parameter's value to *this.color*, and the passed-in *lightsOn* parameter's value to *this.lightsOn*.
* What does this *this* keyword here represent?
* **It's the future object instance of the***Train***class**.
* Essentially, this is all the code that you need to write to achieve two things:
* This code allows me to **build new instances of the***Train***class**.
* Each object instance of the *Train* class that I build will have its own custom properties of *color* and *lightsOn*.
* Now, to actually build a new instance of the *Train* class, I need to use the following syntax:

new Train()

* Inside the parentheses, you need to pass values such as *"red"* and *false*, for example, meaning that the *color* property is set to *"red"* and the *lightsOn* property is set to *false*.
* And, to be able to interact with the new object built this way, you need to assign it to a variable.
* Putting it all together, here's your first train:

var myFirstTrain = new Train('red', false);

* Just like any other variable, you can now, for example, console log the *myFirstTrain* object:

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

* You can continue building instances of the *Train* class. Even if you give them exactly the same properties, they are still separate objects.

var mySecondTrain = new Train('blue', false);

var myThirdTrain = new Train('blue', false);

* However, this is not all that classes can offer.
* You can also add methods to classes, and these methods will then be shared by all future instance objects of my *Train* class.

For example:

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

Now, there are four methods on your *Train* class: *toggleLights()*, *lightsStatus()*, *getSelf()* and *getPrototype()*.

1. The *toggleLights* method uses the logical not operator, *!*. This operator will change the value stored in the *lightsOn* property of the future instance object of the *Train* class; hence the *!this.lightsOn*. And the *=* operator to its left means that it will get assigned to *this.lightsOn*, meaning that it will become the new value of the *lightsOn* property on that given instance object.
2. The *lightsStatus()* method on the *Train* class just reports the current status of the *lightsOn* variable of a given object instance.
3. The *getSelf()* method prints out the properties on the object instance it is called on.
4. The *getPrototype()* console logs the prototype of the object instance of the *Train* class. The prototype holds all the properties shared by all the object instances of the *Train* class. To get the prototype, you'll be using JavaScript's built-in *Object.getPrototypeOf()* method, and passing it *this* object - meaning, the object instance inside of which this method is invoked.

Now you can build a brand new train using this updated *Train* class:

var train4 = new Train('red', false);

And now, you can run each of its methods, one after the other, to confirm their behavior:

train4.toggleLights(); // undefined

train4.lightsStatus(); // Lights on? true

train4.getSelf(); // Train {color: 'red', lightsOn: true}

train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}

* The result of calling *toggleLights()* is the change of true to false and vice-versa, for the *lightsOn* property.
* The result of calling *lightsStatus()* is the console logging of the value of the *lightsOn* property.
* The result of calling *getSelf()* is the console logging the entire object instance in which the *getSelf()* method is called. In this case, the returned object is the *train4* object. Notice that this object gets returned only with the properties ("data") that was build using the *constructor()* function of the *Train* class. That's because all the methods on the *Train* class do not "live" on any of the instance objects of the *Train* class - instead, they live on the prototype, as will be confirmed in the next paragraph.
* Finally, the result of calling the *getPrototype()* method is the console logging of all the properties on the *prototype*. When the *class* syntax is used in JavaScript, this results in **only shared methods being stored on the prototype**, while the *constructor()* function sets up the mechanism for saving instance-specific values ("data") at the time of object instantiation.
* Thus, in conclusion, the class syntax in JavaScript allows us to clearly separate individual object's data - which exists on the object instance itself - from the shared object's functionality (methods), which exist on the prototype and are shared by all object instances.
* However, this is not the whole story.
* It is possible to implement polymorphism using classes in JavaScript, by inheriting from the base class and then overriding the inherited behavior. To understand how this works, it is best to use an example.
* In the code that follows, you will observe another class being coded, which is named *HighSpeedTrain* and inherits from the *Train* class.
* This makes the *Train* class a base class, or the super-class of the *HighSpeedTrain* class. Put differently, the *HighSpeedTrain* class becomes the sub-class of the *Train* class, because it inherits from it.
* To inherit from one class to a new sub-class, JavaScript provides the *extends* keyword, which works as follows:

class HighSpeedTrain extends Train {

}

As in the example above, the sub-class syntax is consistent with how the base class is defined in JavaScript. The only addition here is the *extends* keyword, and the name of the class from which the sub-class inherits.

Now you can describe how the *HighSpeedTrain* works. Again, you can start by defining its constructor function:

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

}

* Notice the slight difference in syntax in the constructor of the *HighSpeedTrain* class, namely the use of the *super* keyword.
* In JavaScript classes, *super* is used to specify what property gets inherited from the super-class in the sub-class.
* In this case, I choose to inherit both the properties from the *Train* super-class in the *HighSpeedTrain* sub-class.
* These properties are *color* and *lightsOn*.
* Next, you add the additional properties of the HighSpeedTrain class inside its constructor, namely, the passengers and highSpeedOn properties.
* Next, inside the constructor body, you use the *super* keyword and pass in the inherited *color* and *lightsOn* properties that come from the *Train* class. On subsequent lines you assign *passengers* to *this.passengers*, and *highSpeedOn* to *this.highSpeedOn*.
* Notice that in addition to the inherited properties, you also **automatically inherit** all the methods that exist on the *Train* prototype, namely, the *toggleLights()*, *lightsStatus()*, *getSelf()*, and *getPrototype()* methods.
* Now let's add another method that will be specific to the *HighSpeedTrain* class: the *toggleHighSpeed()* method.

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

}

Additionally, imagine you realized that you don't like how the *toggleLights()* method from the super-class works, and you want to implement it a bit differently in the sub-class. You can add it inside the *HighSpeedTrain* class.

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLigths();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}

So, how did you override the behavior of the original *toggleLights()* method?

Well in the super-class, the *toggleLights()* method was defined as follows:

toggleLights() {

    this.lightsOn = !this.lightsOn;

}

* You realized that the *HighSpeedTrain* method should reuse the existing behavior of the original *toggleLights()* method, and so you used the *super.toggleLights()* syntax to inherit the entire super-class' method.
* Next, you also inherit the behavior of the super-class' *lightsStatus()* method - because you realize that you want to have the updated status of the *lightsOn* property logged to the console, whenever you invoke the *toggleLights()* method in the sub-class.
* Finally, you also add the third line in the re-implemented *toggleLights()* method, namely:

console.log('Lights are 100% operational.');

* You've added this third line to show that I can combine the "borrowed" method code from the super-class with your own custom code in the sub-class.
* Now you're ready to build some train objects.

var train5 = new Train('blue', false);

var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);

* You've built the *train5* object of the *Train* class, and set its *color* to *"blue"* and its *lightsOn* to *false*.
* Next, you've built the *highSpeed1* object to the *HighSpeedTrain* class, setting *passengers* to *200*, *highSpeedOn* to *false*, *color* to *"green"*, and lightsOn to false.
* Now you can test the behavior of *train5*, by calling, for example, the *toggleLights()* method, then the *lightsStatus()* method:

train5.toggleLights(); // undefined

train5.lightsStatus(); // Lights on? true

* Here's the entire completed code for this lesson:

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLights();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}

var myFirstTrain = new Train('red', false);

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

var mySecondTrain = new Train('blue', false);

var myThirdTrain = new Train('blue', false);

var train4 = new Train('red', false);

train4.toggleLights(); // undefined

train4.lightsStatus(); // Lights on? true

train4.getSelf(); // Train {color: 'red', lightsOn: true}

train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}

var train5 = new Train('blue', false);

var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);

train5.toggleLights(); // undefined

train5.lightsStatus(); // Lights on? true

highSpeed1.toggleLights(); // Lights on? true, Lights are 100% operational.

* Notice how the *toggleLights()* method behaves differently on the *HighSpeedTrain* class than it does on the *Train* class.
* Additionally, it helps to visualize what is happening by getting the prototype of both the *train5* and the *highSpeed1* trains:

train5.getPrototype(); // {constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}

highSpeed1.getPrototype(); // Train {constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}

The returned values in this case might initially seem a bit tricky to comprehend, but actually, it is quite simple:

1. The prototype object of the *train5* object was created when you defined the class *Train*. You can access the prototype using *Train.prototype* syntax and get the prototype object back.
2. The prototype object of the *highSpeed1* object is this object: *{constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}*. In turn this object has its own prototype, which can be found using the following syntax: *HighSpeedTrain.prototype.\_\_proto\_\_*. Running this code returns: *{constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}*.

* Prototypes seem easy to grasp at a certain level, but it's easy to get lost in the complexity. This is one of the reasons why class syntax in JavaScript improves your developer experience, by making it easier to reason about the relationships between classes. However, as you improve your skills, you should always strive to understand your tools better, and this includes prototypes. After all, JavaScript is just a tool, and you've now "peeked behind the curtain".
* In this reading, you've learned the very essence of how OOP with classes works in JavaScript. However, this is not all.
* In the lesson on designing an object-oriented program, you'll learn some more useful concepts. These mostly have to do with coding your classes so that it's even easier to create object instances of those classes in JavaScript.

**Using class instance as another class' constructor's property**

* Consider the following example:

class StationaryBike {

    constructor(position, gears) {

        this.position = position

        this.gears = gears

    }

}

class Treadmill {

    constructor(position, modes) {

        this.position = position

        this.modes = modes

    }

}

class Gym {

    constructor(openHrs, stationaryBikePos, treadmillPos) {

        this.openHrs = openHrs

        this.stationaryBike = new StationaryBike(stationaryBikePos, 8)

        this.treadmill = new Treadmill(treadmillPos, 5)

    }

}

var boxingGym = new Gym("7-22", "right corner", "left corner")

console.log(boxingGym.openHrs) //

console.log(boxingGym.stationaryBike) //

console.log(boxingGym.treadmill) //

* In this example, there are three classes defined: *StationaryBike*, *Treadmill*, and *Gym*.
* The *StationaryBike* class is coded so that its future object instance will have the *position* and *gears* properties. The *position* property describes where the stationary bike will be placed inside the gym, and the *gears* propery gives the number of gears that that stationary bike should have.
* The *Treadmill* class also has a position, and another property, named *modes* (as in "exercise modes").
* The *Gym* class has three parameters in its constructor function: *openHrs*, *stationaryBikePos*, *treadmillPos*.

This code allows me to instantiate a new instance object of the *Gym* class, and then when I inspect it, I get the following information:

* the *openHrs* property is equal to *"7-22"* (that is, 7am to 10pm)
* the *stationaryBike* property is an object of the *StationaryBike* type, containing two properties: *position* and *gears*
* the *treadmill* property is an object of the *Treadmill* type, containing two properties: *position* and *modes*

**Default Parameters**

* A useful a ES6 feature allows me to set a default parameter inside a function definition First, .
* What that means is, I'll use an ES6 feature which allows me to set a default parameter inside a function definition, which goes hand in hand with the defensive coding approach, while requiring almost no effort to implement.

For example, consider a function declaration without default parameters set:

function noDefaultParams(number) {

    console.log('Result:', number \* number)

}

* Obviously, the *noDefaultParams* function should return whatever number it receives, *squared*.

However, what if I call it like this:

noDefaultParams(); // Result: NaN

* JavaScript, due to its dynamic nature, doesn't throw an error, but it does return a non-sensical output.

Consider now, the following improvement, using default parameters:

function withDefaultParams(number = 10) {

    console.log('Result:', number \* number)

}

* Default params allow me to build a function that will run with default argument values even if I don't pass it any arguments, while still being flexible enough to allow me to pass custom argument values and deal with them accordingly.
* This now allows me to code my classes in a way that will promote easier object instantiation.
* Consider the following class definition:

class NoDefaultParams {

    constructor(num1, num2, num3, string1, bool1) {

        this.num1 = num1;

        this.num2 = num2;

        this.num3 = num3;

        this.string1 = string1;

        this.bool1 = bool1;

    }

    calculate() {

        if(this.bool1) {

            console.log(this.string1, this.num1 + this.num2 + this.num3);

            return;

        }

        return "The value of bool1 is incorrect"

    }

}

* Now I'll instantiate an object of the *NoDefaultParams* class, and run the *calculate()* method on it. Obviously, the *bool1* should be set to *true* on invocation to make this work, but I'll set it to false on purpose, to highlight the point I'm making.

var fail = new NoDefaultParams(1,2,3,false);

fail.calculate(); // 'The value of bool1 is incorrect'

* This example might highlight the reason sometimes weird error messages appear when some software is used - perhaps the developers just didn't have enough time to build it better.
* However, now that you know about default parameters, this example can be improved as follows:

class WithDefaultParams {

    constructor(num1 = 1, num2 = 2, num3 = 3, string1 = "Result:", bool1 = true) {

        this.num1 = num1;

        this.num2 = num2;

        this.num3 = num3;

        this.string1 = string1;

        this.bool1 = bool1;

    }

    calculate() {

        if(this.bool1) {

            console.log(this.string1, this.num1 + this.num2 + this.num3);

            return;

        }

        return "The value of bool1 is incorrect"

    }

}

var better = new WithDefaultParams();

better.calculate(); // Result: 6

* This approach improves the developer experience of my code, because I no longer have to worry about feeding the *WithDefaultParameters* class with all the arguments. For quick tests, this is great, because I no longer need to worry about passing the proper arguments.
* Additionally, this approach really shines when building inheritance hierarchies using classes, as it makes it possible to provide only the custom properties in the sub-class, while still accepting all the default parameters from the super-class constructor.

In conclusion, in this reading I've covered the following:

* How to approach designing an object-oriented program in JavaScript
* The role of the *extends* and *super* keywords
* The importance of using default parameters.

**Designing an OO Program**

* In this reading, I will show you how to create classes in JavaScript, using all the concepts you've learned so far.
* Specifically, I'm preparing to build the following inheritance hierarchy:

Diagram

Description automatically generated

* There are two keywords that are essential for OOP with classes in JavaScript.
* These keywords are *extends* and *super*.
* The *extends* keyword allows me to inherit from an existing class.
* Based on the above hierarchy, I can code the *Animal* class like this:

class Animal {

    // ... class code here ...

}

* Then I can code, for example, the *Cat* sub-class, like this:

class Cat extends Animal {

    // ... class code here ...

}

* This is how the *extends* keyword is used to setup inheritance relationships.
* The *super* keyword allows me to "borrow" functionality from a super-class, in a sub-class. The exact dynamics of how this works will be covered later on in this lesson.
* Now I can start thinking about how to implement my OOP class hierarchy.
* Before I even begin, I need to think about things like: \* What should go into the base class of *Animal*? In other words, what will all the sub-classes inherit from the base class? \* What are the specific properties and methods that separate each class from others? \* Generally, how will my classes relate to one another?
* Once I've thought it through, I can build my classes.

So, my plan is as follows:

1. The *Animal* class' constructor will have two properties: *color* and *energy*

2. The *Animal* class' prototype will have three methods: *isActive()*, *sleep()*, and *getColor()*.

3. The *isActive()* method, whenever ran, will lower the value of *energy* until it hits *0*. The *isActive()* method will also report the updated value of *energy*. If *energy* is at zero, the animal object will immediately go to sleep, by invoking the *sleep()* method based on the said condition.

4. The *getColor()* method will just console log the value in the *color* property.

5. The *Cat* class will inherit from *Animal*, with the additional *sound*, *canJumpHigh*, and *canClimbTrees* properties specific to the *Cat* class. It will also have its own *makeSound()* method.

6. The *Bird* class will also inherit from *Animal*, but is own specific properties will be quite different from *Cat*. Namely, the *Bird* class will have the *sound* and the *canFly* properties, and the *makeSound* method too.

7. The *HouseCat* class will extend the *Cat* class, and it will have its own *houseCatSound* as its special property. Additionally, it will override the *makeSound()* method from the *Cat* class, but it will do so in an interesting way. If the *makeSound()* method, on invocation, receives a single *option* argument - set to *true*, then it will run *super.makeSound()* - in other words, run the code from the parent class (*Cat*) with the addition of running the *console.log(this.houseCatSound)*. Effectively, this means that the *makeSound()* method on the *HouseCat* class' instance object will have two separate behaviors, based on whether we pass it *true* or *false*.

8. The *Tiger* class will also inherit from *Cat*, and it will come with its own *tigerSound* property, while the rest of the behavior will be pretty much the same as in the *HouseCat* class.

9. Finally, the *Parrot* class will extend the *Bird* class, with its own *canTalk* property, and its own *makeSound()* method, working with two conditionals: one that checks if the value of *true* was passed to *makeSound* during invocation, and another that checks the value stored inside *this.canTalk* property.

Now that I have fully explained how all the code in my class hierarchy should work I might start implementing it by adding all the requirements as comments inside the code structure.

At this stage, with all the requirements written down as comments, my code should be as follows:

class Animal {

    // constructor: color, energy

    // isActive()

        // if energy > 0, energy -=20, console log energy

        // else if energy <= 0, sleep()

    // sleep()

        // energy += 20

        // console.log energy

}

class Cat extends Animal {

    // constructor: sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound()

        // console.log sound

}

class Bird extends Animal {

    // constructor: sound, canFly, color, energy

    // makeSound()

        // console.log sound

}

class HouseCat extends Cat {

    // constructor: houseCatSound, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // console.log(houseCatSound)

}

class Tiger extends Cat {

    // constructor: tigerSound, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // console.log(tigerSound)

}

class Parrot extends Bird {

    // constructor: canTalk, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // if (canTalk)

            // console.log("talking!")

}

Now that I've coded my requirements inside comments of otherwise empty classes, I can start coding each class as per my specifications.

**Coding the *Animal* class**

First, I'll code the base *Animal* class.

class Animal {

    constructor(color = 'yellow', energy = 100) {

        this.color = color;

        this.energy = energy;

    }

    isActive() {

        if(this.energy > 0) {

            this.energy -= 20;

            console.log('Energy is decreasing, currently at:', this.energy)

        } else if(this.energy == 0){

            this.sleep();

        }

    }

    sleep() {

        this.energy += 20;

        console.log('Energy is increasing, currently at:', this.energy)

    }

    getColor() {

        console.log(this.color)

    }

}

Each animal object, no matter which one it is, will share the properties of *color* and *energy*.

Now I can code the *Cat* and *Bird* classes:

class Cat extends Animal {

    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canClimbTrees = canClimbTrees;

        this.canJumpHigh = canJumpHigh;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

    makeSound() {

        console.log(this.sound);

    }

}

Note: If I didn't use the *super* keyword in our sub-classes, once I'd run the above code, I'd get the following error: *Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor.*

And now I can code the three remaining classes: *HouseCat*, *Tiger*, and *Parrot*.

class HouseCat extends Cat {

    constructor(houseCatSound = "meow", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.houseCatSound = houseCatSound;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        console.log(this.houseCatSound);

    }

}

class Tiger extends Cat {

    constructor(tigerSound = "Roar!", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.tigerSound = tigerSound;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        console.log(this.tigerSound);

    }

}

class Parrot extends Bird {

    constructor(canTalk = false, sound,canFly, color,energy) {

        super(sound,canFly, color,energy);

        this.canTalk = canTalk;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        if (this.canTalk) {

            console.log("I'm a talking parrot!");

        }

    }

}

Now that we've set up this entire inheritance structure, we can build various animal objects.

For example, I can build two parrots: one that can talk, and the other that can't.

var polly = new Parrot(true); // we're passing `true` to the constructor so that polly can talk

var fiji = new Parrot(false); // we're passing `false` to the constructor so that fiji can't talk

polly.makeSound(); // 'chirp', 'I'm a talking parrot!'

fiji.makeSound(); // 'chirp'

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); // setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek', canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180

Also, I can build a pet cat:

var leo = new HouseCat();

Now I can have *leo* purr:

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

Additionally, I can build a tiger:

var cuddles = new Tiger();

My *cuddles* tiger can purr and roar, or just roar:

cuddles.makeSound(false); // Roar!

cuddels.makeSound(true); // purr, Roar!

Here's the complete code from this lesson, for easier copy-pasting:

class Animal {

    constructor(color = 'yellow', energy = 100) {

        this.color = color;

        this.energy = energy;

    }

    isActive() {

        if(this.energy > 0) {

            this.energy -= 20;

            console.log('Energy is decreasing, currently at:', this.energy)

        } else if(this.energy == 0){

            this.sleep();

        }

    }

    sleep() {

        this.energy += 20;

        console.log('Energy is increasing, currently at:', this.energy)

    }

    getColor() {

        console.log(this.color)

    }

}

class Cat extends Animal {

    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canClimbTrees = canClimbTrees;

        this.canJumpHigh = canJumpHigh;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class HouseCat extends Cat {

    constructor(houseCatSound = "meow", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.houseCatSound = houseCatSound;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        console.log(this.houseCatSound);

    }

}

class Tiger extends Cat {

    constructor(tigerSound = "Roar!", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.tigerSound = tigerSound;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        console.log(this.tigerSound);

    }

}

class Parrot extends Bird {

    constructor(canTalk = false, sound,canFly, color,energy) {

        super(sound,canFly, color,energy);

        this.canTalk = canTalk;

    }

    makeSound(option) {

        if (option) {

            super.makeSound();

        }

        if (this.canTalk) {

            console.log("I'm a talking parrot!");

        }

    }

}

var fiji = new Parrot(false); // we're passing `false` to the constructor so that fiji can't talk

var polly = new Parrot(true); // we're passing `true` to the constructor so that polly can talk

fiji.makeSound(); // undefined

fiji.makeSound(true); // chirp

polly.makeSound(); // I'm a talking parrot!

polly.makeSound(true); // chirp, I'm a talking parrot!

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); // setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek', canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180

var leo = new HouseCat();

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

var cuddles = new Tiger();

cuddles.makeSound(false); // Roar!

cuddles.makeSound(true); // purr, Roar!

**Lab: Object Oriented Programming**

// Task 1: Code a Person class

class Person{

    constructor(name = "Tom", age = 20, energy = 100) {

        this.name = name;

        this.age = age;

        this.energy = energy;

    }

    sleep() {

        this.energy += 10;

    }

    doSomenthingFun() {

        this.energy -= 10;

    }

}

// Task 2: Code a Worker class

class Worker extends Person{

    constructor(name, age, energy, xp = 0, hourlyWage = 10) {

        super(name, age, energy);

        this.xp = xp;

        this.hourlyWage = hourlyWage;

    }

    goToWork() {

        this.xp += 10;

    }

}

// Task 3: Code an intern object, run methods

function intern() {

    let intern1 = new Worker("Bob", 21, 110, 0, 10);

    intern1.goToWork();

    return intern1;

}

// Task 4: Code a manager object, methods

function manager() {

    let manager1 = new Worker("Alice", 30, 120, 100, 30);

    manager1.doSomenthingFun();

    return manager1;

}

**Additional resources**

Here is a list of resources that may be helpful as you continue your learning journey.

[Constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor)

[Classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)

[Object-oriented programming](https://css-tricks.com/the-flavors-of-object-oriented-programming-in-javascript/)

[Regular Expressions in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions)

[RegExp object in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp)

**De-structuring arrays and objects**

Have you ever needed to apply the formatting of text in a word processor

from one portion of text to another.

In essence you are copying the properties from some text that is already

formatted and applying them to another piece of text that you want to format.

In a similar way as with text,

JavaScript objects can have properties that define their characteristics.

In this video, you will learn how to D structure, objects and

arrays and also how to use the D structuring syntax,

to extract new variables from objects and arrays.

To illustrate D structuring, imagine that an object or an array is

like a project folder that you have on your computer with several files in it.

D structuring something out of an object or array, in this case your project folder

is like copying that item from your folder on to another location.

The original item still exists in your project folder.

I just made a copy of the original item but

this copy is completely independent of the original item.

Now let's explore an example with an existing built in math object to

D Structure the value of phi from it.

Let's start by using the LET keyword and

surround the uppercase Pie in curly brackets equals math because I

already know that the pie property exists on the math object.

I make a copy of it and I save the new object in a separate variable.

I name Pie.

Note that I can only destruction something that already exists on

an object using faulty spelling, including lower case won't work and

will return an undefined value If I type let open curly bracket small

case pi close curly bracket equals math pi returns as undefined.

This is because there isn't a lowercase pi property on the math object.

So when I try to destroy culture it, I get the value of undefined for

the lower case pi variable.

The next step is to confirm that all caps pi variable has the identical value and

data type as math dot pi by using the triple equal strict comparison operator.

This will return a value of true to prove that the D structured property and

the original property are in no way connected.

I update the value of the pie variable to be one.

By Typing Pi equals one.

I can now compare all caps pie and math dot pi again by typing pie,

triple equal sign, math dot pi, thus getting back the boolean value of false.

It's clear that these two are no longer the same.

The only reason why this worked is because the original property on the object and

the D structured value are not connected in any way.

In other words, there's no connection between the d structured variable and

the source property on the given object.

In this video, you covered how to de structure objects and erase and also how

to use the D structuring syntax to extract new variables from objects and arrays

**For of loops and objects**

In this reading, you'll learn how the for of loop works conceptually.

To begin, it's important to know that a for of loop cannot work on an object directly, since **an object is not iterable**. For example:

const car = {

    speed: 100,

    color: "blue"

}

for(prop of car) {

    console.log(prop)

}

Running the above code snippet will throw the following error:

Uncaught TypeError: car is not iterable

Contrary to objects, arrays *are* iterable. For example:

const colors = ['red','orange','yellow']

for (var color of colors) {

    console.log(color);

}

This time, the output is as follows:

red

orange

yellow

Luckily, you can use the fact that a for of loop can be run on arrays *to loop over objects*.

But how?

Before you can properly answer this question, you first need to review three built-in methods: *Object.keys()*, *Object.values()*, and *Object.entries()*.

**Built-in methods**

**The *Object.keys()* method**

The *Object.keys()* method receives an object as its parameter. Remember, this object is **the object you want to loop over**. It's still too early to explain how you'll loop over the object itself; for now, focus on the returned array of properties when you call the *Object.keys()* method.

Here's an example of running the *Object.keys()* method on a brand new *car2* object:

const car2 = {

    speed: 200,

    color: "red"

}

console.log(Object.keys(car2)); // ['speed','color']

So, when I run *Object.keys()* and pass it my *car2* object, **the returned value is an array of strings**, where each string is a property key of the properties contained in my *car2* object.

**The *Object.values()* method**

Another useful method is *Object.values()*:

const car3 = {

    speed: 300,

    color: "yellow"

}

console.log(Object.values(car3)); // [300, 'yellow']

**The *Object.entries()* method**

Finally, there's another useful method, *Object.entries()*, which returns an array listing both the keys and the values.

const car4 = {

    speed: 400,

    color: 'magenta'

}

console.log(Object.entries(car4));

What gets returned from the invocation of the *Object.entries()* method is the following:

[ ['speed', 400], ['color', 'magenta'] ]

This time, the values that get returned are 2-member arrays nested inside an array. In other words, you get an array of arrays, where each array item has two members, the first being a property's key, and the second being a property's value.

Effectively, it's as if you was listing all of a given object's properties, a bit like this:

[

    [propertyKey, propertyVal],

    [propertyKey, propertyVal],

    ...etc

]

To summarise, you learned that you can loop over arrays using the *for of* loop. You also learned that you can extract object's keys, values, or both, using the *Object.keys()*, *Object.values()* and *Object.entries()* syntax.

**Examples**

You now have all the ingredients that you need to **loop over any object's own property keys and values**.

Here's a very simple example of doing just that:

var clothingItem = {

    price: 50,

    color: 'beige',

    material: 'cotton',

    season: 'autumn'

}

for( key of Object.keys(clothingItem) ) {

    console.log(keys, ":", clothingItem[key])

}

The trickiest part to understand in this syntax is probably the *clothingItem[key]*.

Luckily, this is not too hard to comprehend, especially since you've already covered the concept previously when you were learning **how to access an object's member using the brackets notation**.

Recall that you also learned how you can dynamically access a property name.

To revisit this concept and show a practical demo of how that works, let's code a function declaration that randomly assigns either the string *speed* or the string *color* to a variable name, and then build an object that has only two keys: a *speed* key and a *color* key.

After this setup, you will be able to dynamically access either one of those properties on a brand new *drone* object, using the brackets notation.

Here's the example's code:

function testBracketsDynamicAccess() {

  var dynamicKey = Math.random() > 0.5 ? "speed" : "color";

    var drone = {

      speed: 15,

      color: "orange"

    }

    console.log(drone[dynamicKey]);

}

testBracketsDynamicAccess();

This example might feel a bit convoluted, but its purpose is to demo the fact that you're getting either one or the other value from an object's key, based on the string that got assigned to the *dynamicKey* variable, and accessed without issues, using the brackets notation.

Feel free to run the *testBracketsDynamicAccess()* function a few times, and you'll notice that sometimes the value that gets output is *15*, and sometimes it's *orange*, although I'm always accessing the *drone[dynamicKey]* key. Since the *dynamicKey* is an expression that gets evaluated, based on the *Math.random()* invocation, sometimes that expression evaluates to *drone["speed"]*, and other times that expression evaluates to *drone["color"]*.

You have now learned about the building blocks that make the for of loop useful to iterate over objects - *although objects are not iterables.*

Next, you'll have a go at a practical example of working with both the for of and the for in loop. Each loops have their place and can be considered useful in different situations.

**For- of loops and objects**

In this video, I'll help you understand the difference

between for and loops and for of loops,

when applied to objects in JavaScript.

At first, this code may appear complex,

but you'll find that it only involves

concepts that you've dealt with before.

Let's begin by breaking down what I have.

I have a concept to an object

and assign to a variable named car.

This object has three properties, engine,

steering and speed, which are given the values true,

true, and slow respectively.

I also have a cost assigned to the variable sports car,

and I've used Object.create,

so that inherits the properties of the car variable.

Then I set the speed property of the sports car

to fast and save it to the sports car variable.

I've also added a console log to display the string,

the sports car object,

followed by the properties of the sports car object.

Next, I have two for loops.

The first one is a for-in loop written to

log the properties of the sports car object.

This is typed as for,

followed by properties and sports car in parentheses.

Inside of curly braces is a console log for prop,

so it displays the properties of the sports car.

Before and after the foreign loop code,

I've written console logs to print

some comments which will be

addressed after I run my code.

My second for loop uses

the object keys method on my sports car object,

which should iterate over that object.

This is typed as for,

followed by prop of objects,

dot keys, sports car inside of parentheses.

Inside of the curly braces is

a console log for prop plus a colon

in double-quotes and sports car

with prop enclosed in square brackets.

I've also console logged another comma after this code.

When I run this code,

I get the output.

The sports car object is speed fast.

It also displays a comment I had in

the code for in is unreliable.

Let's explore why.

From the foreign loop,

the console log values might surprise us.

It loops over the property in the sports car object,

but the output includes the properties on

the sports car's prototype too.

Instead of just a speed property,

I also looped over the engine and steering

properties which exist on

the prototype of the sports car object,

not on the sports car object itself.

In fact, as the common state,

for in loops are unreliable in this scenario because

they iterate over not only the specified object,

but also its prototype.

That's why I used a thinking emoji in my comment with

a string iterating over objects and its prototype.

Now, let's examine what came from the second for loop,

which is a for of loop.

Note that our output, the comment I had in

the code for of is reliable.

I've gotten back speed fast, or in other words,

the property and value that I've

assigned to the sports car object.

Just as I have stated in the comment in my code.

This is because a for of loop only iterates over

the object's own properties and

does not count the prototype at all.

Hopefully, you are able to differentiate between

the for in and the for of types of loops.

Let's run a simplified version of

the same code without the comments.

Also show my other file.

This time around my car object only has

an undrawn property then I'm declaring

the sports car variable and I'm setting

the car object as the sports car prototype.

I'm also setting the speed of the sports car as fast.

Finally, I'm console

logging the sports car object and looping

over prop and sports car and

props of object keys and sports car.

In summary, the two loops are simplified.

I'm only console logging a thinking emoji and the props

value for the first loop, the foreign loop.

I'm only console logging a bull's eye emoji and

the provenance value in the for of loop.

That is the second loop in my code.

When I run the code again,

the result is much simpler.

First, I confirm that the sports car object only

has one single property speed, which is fast.

Then the for in loop displays speed and engine,

because engine belongs to the prototype of

the sports car object and not

the sports car object itself.

Finally, the for of loop outputs

looping over only the sports car object properties,

which in this case is only the property fast.

The emojis from my code up here alongside

these properties and help us

to understand what they mean.

Just to sum it up, the speed and

engine properties are from the foreign loop

because it draws from the prototype of

the sports car object as well as the object itself.

The speed fast property, however,

comes from the for of loop,

since that iterates over

the sports car objects property only.

In this video, you learn that when you run on

objects in JavaScript for in loops,

iterate over the properties of

the object and its prototype.

While for of loops do this

only for the objects' properties.

# Template literals examples

The aim of this reading is to help you understand how template literals work.

## What are template literals?

Template literals are an alternative way of working with strings, which was introduced in the ES6 addition to the JavaScript language.

Up until ES6, the only way to build strings in JavaScript was to delimit them in either single quotes or double quotes:

'Hello, World!'

"Hello, World!"

Alongside the previous ways to build strings, ES6 introduced the use of backtick characters as delimiters:

`Hello, World!`

The above code snippet is an example of a template string, which is also known as a template literal.

Note: On most keyboards, the backtick character can be located above the TAB key, to the left of the number 1 key.

## Differences between a template and regular string

There are several ways in which a template string is different from a regular string.

* First, it allows for **variable interpolation**:

let greet = "Hello";

let place = "World";

console.log(`${greet} ${place} !`)

The above console log will output:

Hello World !

Essentially, using template literals allows programmers to embed variables directly in between the backticks, without the need to use the + operator and the single or double quotes to delimit string literals from variables. In other words, in ES5, the above example would have to be written as follows:

var greet = "Hello";

var place = "World";

console.log(greet + " " + place + "!");

* Besides variable interpolation, template strings can span multiple lines.

For example, this is perfectly good syntax:

`Hello,

World

!

`

Notice that this can't be done using **string literals** (that is, strings delimited in single or double quotes):

"Hello,

World"

The above code, when run, will throw a syntax error.

Put simply, template literals allow for multi-line strings - something that simply isn't possible with string literals.

* Additionally, the reason why it's possible to interpolate variables in template literals is because this syntax actually allows for **expression evaluation**.

In other words, this:

console.log(`${1 + 1 + 1 + 1 + 1} stars!`)

The above example will console log the following string: 5 stars!.

This opens up a host of possibilities. For example, it's possible to evaluate a ternary expression inside a template literal.

Some additional use cases of template literals are **nested template literals** and **tagged templates**. However, they are a bit more involved and are beyond the scope of this reading.

If you're curious about how they work, please refer to the additional reading provided at the end of this lesson.

# Working with template literals

In this video, I'll guide you through creating and using

template literals in JavaScript ES6

to understand their benefits.

First, let's revisit strings in JavaScript ES5,

which are built using single quotes or double quotes.

I have an example in my code,

I've declared a variable,

no multi-line with the let keyword,

which holds the string value,

no multi-line strings in ES5.

On the following line,

I'm using the console.log method to

output the string did you know,

followed by the plus operator and

then finally the no multi-line variable.

Recall that you can use

the plus operator for concatenation.

When I run the code, I get

the expected output of did you know,

no multi-line strings in ES5.

This time, let's change

the value of the no multi-line variable.

To do this, I place my cursor after the word no

and press the Enter key to move

part of the string to a second line.

This now spans the string over multiple lines.

If we were to run this code now,

we would get the error.

Also, notice that

VS Code is trying to warn us about this error by

highlighting the end of lines 3 and 4 with

red text and displaying

the message unterminated string literal.

This is basically JavaScript

telling us that our single line string is

not coded correctly with

the expected closing double quotation symbol.

Using ES5 methods,

you can only create strings using

single or double quotations and using

this method does not support

the use of multi-line strings.

Next, let me clear the output

and switch tabs in my code.

Here we have some syntax that may be unfamiliar to you.

We have a variable multi-line

declared with the let keyword and it contains a string.

Like the previous example,

this string is distributed across several lines.

However, instead of quotes,

it's encased within a pair of backticks.

This turns it into an ES6 expression

known as a template literal.

The backtick symbol is usually located above

the tab key to the left of the one key on your keyboard.

Still, it's a good idea to check your devices

documentation as you may be

using a different keyboard layout.

Using template literals, you can add

as many lines as you want without causing errors.

I can confirm this by running my code.

Notice that the output is a multi-line string.

But wait, it gets better.

We can combine template literals with

variable interpolation for even more flexibility.

Once again, let me clear

the output and continue to the next tab.

Here, notice I have

two variables called first and second,

each containing a single line string

encased in backticks.

But there's a twist here.

Each string also has

a segment that sits inside of quotes.

Wouldn't this cause conflict you might wonder?

Fortunately, that's not the case.

By using template literals,

JavaScript does not consider quotes,

a string delimiters,

meaning they are just regular characters here.

Now I can use template literals again to interpolate

the variables first and second on a different line.

Instead of using the plus operator for concatenation,

I can simply enclose

the full desired string within backticks.

I can then place the variable name by enclosing it within

a set of curly braces preceded by a dollar symbol.

If I run this code,

JavaScript combines everything and

outputs the full string.

With template literals you don't need to worry about

the limitations of using single and double quotes.

This can make for a much better coding experience

as you just need to use

backticks and variable interpolation.

That's all for this video.

Well done, you learned how to

build then apply template literals to

create multiline strings and

interpolate variables in JavaScript.

**Lab: Advanced JS Features**

// Task 1

var dairy = ['cheese', 'sour cream', 'milk', 'yogurt', 'ice cream', 'milkshake'];

function logDairy() {

    for (key of dairy) {

        console.log(key);

    }

}

logDairy();

// Task 2

const animal = {

    canJump: true

};

const bird = Object.create(animal);

bird.canFly = true;

bird.hasFeathers = true;

function birdCan() {

    for (key of Object.keys(bird)) {

        console.log(key,": ",bird[key]);

    }

}

birdCan();

// Task 3

function animalCan() {

    for (key in bird) {

        console.log(key,": ",bird[key]);

    }

}

animalCan();

# Data Structures

Suppose you receive some data on students test results and your task is to write

a program that outputs an average grade from all the tests based on the raw data.

Before you can code this task, you need to consider two separate issues.

First, how do you represent the given data in your app and

second, how do you called the solution?

Before you even start coding a solution,

you need to think about how you will represent the data.

In this video, you will learn about some of the JavaScript's most

common data structures such as objects, arrays, maps and sets.

A data structure is a way to organize data.

For example, you could represent it as a string.

However, it would be somewhat of a strange representation.

You'd need to somehow extract the numbers from the string before performing

calculations on them.

Alternatively, you could represent it as several numbers,

each number saved in a variable.

Doing it this way you don't have to extract and convert strings to numbers.

But is this the most efficient way of storing your apps data?

Perhaps another approach would be to store all the grades in an array.

This way of organizing your data is even more efficient.

It involves less typing and

we're grouping related data together under a single label.

You may recall the many benefits of using a arrays.

But what is important here is that you understand how you code

a solution to a given task depends on how you structure your data.

In other words,

a solution to a coding task depends on the data structure you use.

JavaScript is somewhat limited in the types of data structures available

compared to other programming languages, such as, Java or Python.

However, some of the most common that you will encounter are objects arrays,

maps and sets.

You may be familiar with some of these already.

Let's explore each briefly.

Now you may recall that an object is unaltered,

noniterable collection of key value pairs and

you use objects when you need to store and later access a value under a key.

An example of using this data structure is when you need to write

object orientated, R00P code.

Similarly, you may also recall an array which is an ordered iterable

collection of values.

Likewise, you use arrays when you need to store and

later access a value under an index.

And remember, we do not specify the index, JavaScript does this automatically.

You only use the index to access the specific value stored in the array.

When working with arrays, it's common to use a loop, such as a for

loop to access and edit the data.

For example, to find the average grade for the task earlier,

you could loop over the array and calculate the total sum.

Then after the for loop,

calculate the average by dividing the sum by the length of the array.

The next data structure is map which is like an array because it's iterable.

However, it consists of key value pairs.

It's important not to confuse a map with an object.

With maps any value can be used as a key.

With objects, keys can only be strings or symbols.

Finally, the last data structure I want you to know about is a set.

This is another collection where each item in the collection must be unique.

For example, if you try to add a non unique item to a set,

this operation will simply not be run.

In other words, no errors will be thrown and no updates will be made to a set.

In this video, you learned about some of JavaScript's most common data structures,

such as, objects, arrays, maps and

sets, which one you need to use depends on the task at hand.

Next time you have a coding task,

try to think of the data structures you might use.

# Data Structures examples

In this reading, you'll learn about some of the most common examples of data structures.

The focus will be on working with the Object, Array, Map and Set data structures in JavaScript, through a series of examples.

Note that this reading will not include a discussion of some data structures that exist in some other languages, such as Queues or Linked Lists. Although these data structures (and other data structures too!) can be custom-coded in JavaScript, the advanced nature of these topics and the difficulty with implementing related exercises means they are beyond the scope of this reading.

## Working with arrays in JavaScript

Previously, you've covered a lot of concepts related to how to work with JavaScript arrays.

However, there are still a few important topics that can be covered, and one of those is, for example, working with some built-in methods.

In this reading, the focus is on three specific methods that exist on arrays:

1. forEach
2. filter
3. map

Let's explore these methods.

### The forEach() method

Arrays in JavaScript come with a handy method that allows you to loop over each of their members.

Here's the basic syntax:

const fruits = ['kiwi','mango','apple','pear'];

function appendIndex(fruit, index) {

    console.log(`${index}. ${fruit}`)

}

fruits.forEach(appendIndex);

The result of running the above code is this:

0. kiwi

1. mango

2. apple

3. pear

To explain the syntax, the forEach() method accepts **a function that will work on each array item**. That function's first parameter is the current array item itself, and the second (optional) parameter is the index.

Very often, the function that the forEach() method needs to use is passed in directly into the method call, like this:

const veggies = ['onion', 'garlic', 'potato'];

veggies.forEach( function(veggie, index) {

    console.log(`${index}. ${fruit}`);

});

This makes for more compact code, but perhaps somewhat harder to read. To increase readability, sometimes arrow functions are used. You can find out more about arrow functions in the additional reading.

### The filter() method

Another very useful method on the array is the filter() method. It filters your arrays **based on a specific test**. Those array items that pass the test are returned.

Here's an example:

const nums = [0,10,20,30,40,50];

nums.filter( function(num) {

    return num > 20;

})

Here's the returned array value:

[30,40,50]

Similar to the forEach() method, the filter() method also accepts a function and that function performs some work on each of the items in the array.

### The map method

Finally, there's a very useful map method.

This method is used to map each array item over to another array's item, based on whatever work is performed inside the function that is passed-in to the map as a parameter.

[0,10,20,30,40,50].map( function(num) {

    return num / 10

})

The return value from the above code is:

[0,1,2,3,4,5]

As already discussed, choosing a proper data structure affects the very code that you can write. This is because the data structure itself comes with some built-in functionality that makes it easier to perform certain tasks or makes it harder or even impossible without converting your code to a proper data structure.

Now that you've covered the methods, let's explore how to work with different built-in data structures in JavaScript.

## Working with Objects in JavaScript

A lot of the information on how to work with objects in JavaScript has already been covered in this course.

The example below demonstrates how to use the object data structure to complete a specific task. This task is to convert an object to an array:

const result = [];

const drone = {

    speed: 100,

    color: 'yellow'

}

const droneKeys = Object.keys(drone);

droneKeys.forEach( function(key) {

    result.push(key, drone[key])

})

console.log(result)

This is the result of executing the above code:

['speed',100,'color','yellow']

Although this is possible and works, having to do something like this might mean that you haven't chosen the correct data structure to work with in your code.

On the flip side, sometimes you don't get to pick the data structure you're working with. Perhaps that data comes in from a third-party data provider and all you can do is code your program so that it consumes it. You'll learn more about the interchange of data on the web when you learn about JSON (JavaScript Object Notation).

## Working with Maps in JavaScript

To make a new Map, you can use the Map constructor:

new Map();

A map can feel very similar to an object in JS.

However, it doesn't have inheritance. No prototypes! This makes it useful as a data storage.

For example:

let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.set(2, "The Runner-up");

bestBoxers.set(3, "The third place");

console.log(bestBoxers);

Here's the console output:

Map(3) {1 => 'The Champion', 2 => 'The Runner-up', 3 => 'The third place'}

To get a specific value, you need to use the get() method. For example:

bestBoxers.get(1); // 'The Champion'

## Working with Sets in JavaScript

A set is a collection of unique values.

To build a new set, you can use the Set constructor:

new Set();

The Set constructor can, for example, accept an array.

This means that we can use it to quickly filter an array for unique members.

const repetitiveFruits = ['apple','pear','apple','pear','plum', 'apple'];

const uniqueFruits = new Set(repetitiveFruits);

console.log(uniqueFruits);

The above code outputs the following in the console:

{'apple', 'pear', 'plum'}

## Other data structures in JavaScript

Besides the built-in data structures in JavaScript, it's possible to build non-native, custom data structures.

These data structures come built-in natively in some other programming languages or even those other programming languages don't support them natively.

Some more advanced data structures that have not been covered include:

* Queues
* Linked lists (singly-linked and doubly-linked)
* Trees
* Graphs

For resources on building these data structures, please refer to the additional reading.

# Spread operator

The spread operator is another of

the many great new editions that came to

the JavaScript language in its ES6 update.

It is the shortest and simplest method to copy

the properties of an object onto a newly created object.

Think of the spread operator as

a magical multi-purpose tool that can

spread out array items and join objects together.

In this video, you will explore what

the spread operator is and how to use it.

Note that the spread operator

is characterized by three dots.

To illustrate where the spread operator becomes useful,

let's first build an array and

then call the items with a basic function.

Using the variable keyword let,

I create an array of the top three places to visit on

a holiday to Rome and I call it top3.

Then I include the Colosseum, Trevi Fountain,

and the Vatican City

as the elements of my top three array.

Now I write a function that

lists these three attractions.

To keep it simple, I just log it into the console,

I create a function and I name it,

showItinerary with three arguments,

place1, place2, place3.

Now I console log the three places to visit by

typing console.log and then the string Visit,

and the first argument, place1.

Next, I console log

the other two arguments by first using the strings,

then visit, and the argument place2.

Then the string finish with a visit to,

and the third argument place3.

Then when I run the function

showItinerary it should run through all three places.

Now, let's try to run

the showItinerary function with

the elements in the top3 array.

I write the function name, showItinerary,

and each array element starting with top3,

open square bracket zero,

which is the first element in the array,

and close square bracket.

Then I do the same for

the other array elements one and two.

But what if I extended my function

to have seven places that I want to visit?

Let's disregard what

the showItinerary function definition may show and

imagine that I have made

the necessary updates so that

the function will handle seven arguments.

I have now built an entirely new array

of the top seven destinations,

and I've saved it in a variable named top7.

Now I call the showItinerary function

and passing this function,

the seven arguments again start with zero as

in top7 and zero within square brackets.

Then the array again, top7

and number one within square brackets,

and so on until we get to

the last top seven array element,

which is top7 and number 6 within square brackets.

While this code will work,

it seems a bit impractical

the spread operator simplifies things. Let's try it now.

I type the showItinerary function, then open parenthesis,

three dots, top7,

close parenthesis, semicolon, that's it.

I use the spread operator by placing it in front

of an array which is passed

to the showItinerary function.

The advantage of this approach is

that you don't have to list

each individual member of

the array that you want to pass to your function.

The syntax is clear,

concise, and easy to type.

Now, if I go back to the top3 array,

I can use it with

the spread operator just like

I used it for the top7 array.

I type the showItinerary function and

the three dots and the array name,

top3 within parenthesis and I end with a semicolon.

The function returns the combination of

the strings and the variables I used earlier,

visit the Colosseum, then visit

Trevi Fountain and finish

with a visit to the Vatican City.

In this video, you explored what

the spread operator is and how you can use it.

# Rest operator

You might already know, that a spread operator in JavaScript,

is used to unpack a box, for example, to unpack an array.

The rest operator, on the other hand, is used to build a smaller box, and

pack items into it.

In this video, you will explore what a rest operator is, and how to use it.

To illustrate how a rest operator can be useful, consider a travel itinerary,

I've realized, that there are many things I'd love to see, while in Rome,

during my vacation.

Unfortunately, I'm pressed for time, and

there's no way to really enjoy all of the attractions,

even if I rush to see them all, let's explore how to solve this itinerary issue.

In this case I can create an array of seven places that I would like to visit,

on my holiday to Rome.

So I type const top7 to create the array, and

I list the seven places I would like to go, within square brackets,

with each item in double quotes, and separated by a comma.

The Colosseum, the Roman Forum, the Vatican, Trevi Fountain,

the Pantheon, Piazza Venezia, and the Palatine Hill.

To solve the problem, I decided to organize my list in such a way,

that I have my top three attractions, and prepare another list, for a second visit.

To do this, I use the rest operator, and a technique known as de structuring,

in this de structure in syntax, I simply specify,

what I want to get out of the top seven array, in this case the first three items.

So, I start by typing const, open and closed square bracket, equal sign, top7.

Then, I create three variables first, second and third,

plus another variable, to hold the rest of the list, named secondVisit.

To do this, I type const first, second, third, and

then three dots, and a new variable, secondVisit,

within Square brackets, then equals top7, semi colon.

With this syntax, I extracted the contents of the top7 array,

into four variables, the variable first, with the value of the Colosseum.

The variable second, with the value of the Roman Forum, the variable third,

with the value of the Vatican.

And then I include the variable secondVisit, that is a sub array,

of the rest of the items, in the original top7 array.

If I inspect secondVisit, an array of the four remaining attractions,

Trevi Fountain, the Pantheon, Piazza Venezia, and the Palatine Hill,

are all saved as string primitives.

The rest operator, therefore,

gives us what is left over of the source array, as a separate sub array.

The rest operator is also useful in functions too, in fact,

I can use a rest parameter, to quickly multiply values.

I create a function, to add the tax rate to prices, I name it,

addTaxToPrices, and give it two parameters,

the tax rate and the rest operator, with the items bought.

The function returns each item with the tax rate, in the addToTaxPrices,

parameters, the rest parameter gives me an array, so

I can use array methods, on items bought, using the map method.

It's important to know, that the rest parameter,

must be the last parameter in the function definition.

This means, that adding any other parameter, to my addTaxToPrices function,

after the rest operator, and the items bought would result in an error.

In this video, you learned how to use the rest operator,

which groups the remaining parameters in a list, within a standard JavaScript array,

well done with completing this video, on the rest operator.

# Using Spread and Rest

In this reading, you'll learn how to join arrays, objects using the rest operator. You will also discover how to use the spread operator to:

* Add new members to arrays without using the push() method,
* Convert a string to an array and
* Copy either an object or an array into a separate object

#### Recall that the push() and pop() methods are used to add and remove items from the end of an array.

## Join arrays, objects using the rest operator

Using the spread operator, it's easy to concatenate arrays:

const fruits = ['apple', 'pear', 'plum']

const berries = ['blueberry', 'strawberry']

const fruitsAndBerries = [...fruits, ...berries] // concatenate

console.log(fruitsAndBerries); // outputs a single array

Here's the result:

['apple', 'pear', 'plum', 'blueberry', 'strawberry']

It's also easy to join objects:

const flying = { wings: 2 }

const car = { wheels: 4 }

const flyingCar = {...flying, ...car}

console.log(flyingCar) // {wings: 2, wheels: 4}

## Add new members to arrays without using the push() method

let veggies = ['onion', 'parsley'];

veggies = [...veggies, 'carrot', 'beetroot'];

console.log(veggies);

Here's the output:

['onion', 'parsley', 'carrot', 'beetroot']

## Convert a string to an array using the spread operator

Given a string, it's easy to spread it out into separate array items:

const greeting = "Hello";

const arrayOfChars = [...greeting];

console.log(arrayOfChars); //  ['H', 'e', 'l', 'l', 'o']

## Copy either an object or an array into a separate one

Here's how to copy an object into a completely separate object, using the spread operator.

const car1 = {

    speed: 200,

    color: 'yellow'

}

const car 2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)

The output is 201, 200.

You can copy an array into a completely separate array, also using the spread operator, like this:

const fruits1 = ['apples', 'pears']

const fruits2 = [...fruits1]

fruits1.pop()

console.log(fruits1, "not", fruits2)

This time, the output is:

['apples'] 'not' ['apples','pears']

Note that the spread operator only performs a shallow copy of the source array or object. For more information on this, please refer to the additional reading.

There are many more tricks that you can perform with the spread operator. Some of them are really handy when you start working with a library such as React.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[Template literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

[Arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

[Spread syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)

[Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)

[JavaScript data structures](https://data-flair.training/blogs/javascript-data-structures/)

# JavaScript modules

Hello and welcome.

Photographers often need different camera lenses for different shooting situations.

It wouldn't be very economical to have a separate camera for

every need that arises, fortunately most systems are modular

meaning the lens can be removed and replaced with a different one as needed.

In JavaScript, complex programs can be useful for multiple applications and

it wouldn't make sense to rewrite them over and over thankfully.

Since version ES6, they can be saved and used elsewhere as modules in this video.

You will cover what modules are in JavaScript which includes how they work as

well as their current limitations.

You'll also explore how to use simple ES6 modules in a browser.

JavaScript modules are standalone units of code that you can reuse again and again.

Being standalone means that you can add them to your program, remove them and

replace them with other modules and everything will still work.

But before we continue,

let's explore what things were like before modules entered the picture.

In all versions of JS,

all functions that are defined on the window object are global.

While useful for simple projects, this created some issues

when third party libraries or multiple developers became involved.

For example, a global function from one script could

override a function from another one using the same variable name,

techniques were developed to bypass these issues but they were not without flaws.

And so for a long time,

JavaScript lacked built in natively supported module functionality.

An engineer at Mozilla named Kevin Bangor tried to fix this through

a project called Server JS which was later renamed to Common JS.

CommonJS is designed to specify how modules

should work outside of the browser environment.

It is mostly used on server side JavaScript namely node.js

a downside of CommonJS is that browsers don't understand its syntax.

That is certain keywords that CommonJS relies on, such as require and

module.exports don't work as expected in browsers.

You can learn more about other module systems in the additional reading but

now that you know a bit of the history,

let me demonstrate ES6 modules in action to understand how ES6 modules work.

We need to go back to the humble html script tag.

The script tag comes with a type attribute which identifies the type

of content that it contains.

For JavaScript, it is written as script type equals followed by text slash

JavaScript in double quotes to display text in a browser on the next line.

I've typed console.log, followed by the string hello from script tag

in parenthesis and then a script closing tag.

I can even omit the type attribute and use a basic script tag and

the attribute will be set to text/JavaScript by default.

To make that happen I've input an open script tag followed by

console.log on the next line and same sample text.

I've been added a few more console logs before closing off the script tag.

To verify that these script tags work.

I'll open a local html file in my browser that is linked to this code

when I opened the elements tab in the developer tools,

it displays the script tags while the console tab shows the console log output.

Next let's explore how we can use ES6 modules in a browser before

starting I will open a separate code document.

Now I've changed the value of the type attribute by typing the script tag,

script type equals module.

I then imported a module called greeting.js.

But how do you make it work to demonstrate I'll open the associated

html file in another browser tab.

Since I'm running this file locally, access to my script is blocked by

a built in browser security feature called cross origin resource sharing and

I receive an error message under the console tab.

To eliminate this error, I need to run the html file over a server.

I can use a local server for that.

For information on how to set up a local server referred to the additional reading,

now my greeting.js module is being read in and

consumed by this html document script tag.

In other words, the file script tags have successfully imported the module.

In this video you have learned about modules in JavaScript ES6.

You should now have a better idea of both their uses and their limitations and

you should know how to use basic ES6 modules in a browser.

# JavaScript DOM manipulation

Did you ever think of

the functions your TV remote does for you?

It allows you to make changes to your TV.

You can think of the DOM like a TV remote that

lets you change the webpage content on the screen.

It even allows you to replace

only certain parts of the page,

such as an image or the heading or both.

A remote however, has certain limitations.

You can only change the channel,

contrast, brightness, and volume.

The DOM gives you

much finer-grained control than a TV remote ever could.

For example, imagine you can change what characters in

a TV movie look like with

a few simple commands while the movie is playing.

The DOM allows you to change

properties of objects on a webpage.

You can think of the DOM as

a superpower remote for websites.

It gives developers power in how they

can manipulate and update webpages.

The DOM is in the form of

a JavaScript object with

nested objects for different parts of the page.

These objects have nested objects of their own until

the entire HTML file is

mapped out in what looks like a tree structure.

The DOM is the model of the HTML file

saved as a JavaScript object in your browser's memory.

The browser automatically builds

the DOM for every webpage that it downloads.

For example, if you type

a URL into your browser's address bar,

the browser would fetch

the webpage that exists on this domain.

If the browser connects to the server and

it allows the browser to download the page,

it will receive all the HTML code

and save it in its memory.

The browser will then show the downloaded page.

It would also build that webpage's DOM a model

of the HTML document your browser has just downloaded.

As a developer, you can interact with

the page's DOM to make changes to the webpage.

To interact with the DOM,

you can use the Elements tab

inside the browser's DevTools.

You get to the DevTools by right-clicking anywhere

in a browser window and then by clicking "Inspect".

The Elements tab allows you to interact with the DOM of

the currently active webpage

using a graphical user interface,

also known as a GUI.

The browser also allows you to

interact with the DOM using JavaScript.

To do this, you should click

the Console tab in the browser's DevTools.

You can focus on the Console panel by

pressing the Escape key on your keyboard.

Once done, you can start typing

JavaScript commands to view and manipulate the DOM.

This is similar to how you

interact with the DOM using the Elements panel,

only this time you're using code to do it.

The entire DOM object is saved inside

the document variable and accessible via the console.

Now let me demonstrate how DOM manipulation works.

Using the document object model

allows us to manipulate live websites.

For example, I've opened my browser and loaded

the example.com webpage from the Internet.

I want to add a heading 2 HTML element

to this webpage using the DOM.

Once again, I have the Developer Tools open in

the window to the right on the Console tab is active.

It's important to remember that any changes I make to

the DOM are relative to

my browser's local copy of the webpage.

I'm not updating the content of

the live website so don't worry,

you're not going to break it.

If I reload the webpage,

all changes I make to the DOM will be lost as it

will reset to the page

that was downloaded from the server.

In order to create my heading 2

I need to update the document object.

To do this, I type document

dot and then the method createElement.

Then inside the open and close parenthesis,

I type the name of

the HTML element that I want to create,

which is wrapped around single quotes.

What I am doing here is passing

the tag names as a JavaScript string.

Finally, I'm going to assign

this statement to a const variable.

I type constant h2,

and then the assignment operator, I press "Enter."

Now I have successfully built my h2 element.

You may notice that the webpage on

the left has not been updated with the heading 2.

This is because although,

I now have a h2 element saved in JavaScript's memory,

I still need to attach it to the DOM structure.

Currently, it's not a part of the document object.

Additionally, my h2 element also needs some text.

Without it, even after adding

the h2 element to the document,

there would not be a visible change on

the page because the added h2 element,

although, being a part of the DOM,

would have no text inside.

In other words, it would be blank.

Let's fix these two issues now.

First, I'll make some updates to my h2 element.

I'll add some attributes and

some text using the inner text method.

I can run this on my h2 variable.

I type h2.innerText and then assign

a string with a value of this is an h2 heading.

Next, I want my heading 2 element to have

an ID and a class HTML attribute.

To do this, I need to use the set attribute method.

I type h2.setAttribute,

which takes two parameters.

The first one is the attribute's name,

which is ID,

and then the value of the attribute.

For this example, I'll name as

sub-heading and press "Enter."

Notice that inside the parenthesis,

the name and the value are wrapped in

single quotes and separated by a comma.

Next, I need to use the set attribute method again.

This time for the class attribute.

I've pasted the same line of code,

and now I just need to change the word ID to class.

For the value, I change this

to secondary and press "Enter."

Finally, I'm ready to append my h2 object to the DOM.

First, I can test that my code is correct by

typing h2 and noticing the HTML structure's output.

This looks good to me so let's add it to the DOM.

I need to run the append child method

on the document body to do this.

Now, I type document.body.appendChild.

Inside the parenthesis,

I place my HTML structure stored in the variable h2.

Success. My object is appended to the body of

this webpage and

the heading 2 text displays in the browser.

You should now be able to explain the basics of

DOM manipulation and use some of

the most common DOM manipulation methods.

**Code from lesson**

const h2 = document.createElement('h2');

undefined

h2.innerText = "This is an h2 heading"

"This is an h2 heading"

h2.setAttribute('id', 'sub-heading')

undefined

h2.setAttribute('class', 'secondary')

undefined

h2

<h2 id="sub-heading" class="secondary">

document.body.appendChild(h2)

<h2 id="sub-heading" class="secondary">

# JavaScript interactivity

The purpose of this reading is to introduce you to a simple explanation of web page manipulation and some examples of it.

Did you know that JavaScript's initial purpose was to **provide interactivity in the browser?**

In other words, it was the "set of controls" that would allow web developers to control the behavior of the webpages and even the browsers that these webpages worked on.

This is still the case today.

As the web ecosystem developed and the world became ever more digitized, so did the myriad of ways that one can use JavaScript as a web developer to manipulate websites.

Initially, in the late 1990s, there was plain JavaScript that had to be tweaked to suit individual browsers.

Then, by the mid-2000s, the jQuery library came out, with the idea of writing less code, but doing more with it. It "leveled the playing field" as it allowed developers to use a single code-base for various browsers.

This trend continued and many other frameworks such as React, Vue, Angular, D3, and more came along.

Together with npm and Node.js, the JavaScript ecosystem is not slowing down.

Even though it has gone a long way, sometimes it's good to go back to basics. JavaScript is still the king when it comes to making our websites interactive.

Although CSS has developed significantly over the years, it is still JavaScript that allows users to:

* Get their geolocation,
* Interact with maps,
* Play games in the browser,
* Handle all kinds of user-triggered events, regardless of the device,
* Verify form input before sending it to the backend of a webapp,
* and more!

There are many, many ways in which JavaScript allows you to build rich, interactive experiences on the web.

# JavaScript selectors

One of the most powerful features of JavaScript is dom manipulation.

For example, you can click on a button and change the color of some text or

even display a pop up message.

Essentially, you're dynamically updating the html content in real time.

To do this, you must be able to locate the objects in your html document that you

want to manipulate.

In this video, I'll guide you through how to use selectors in

JavaScript to quickly find specific objects in a document.

JavaScript selectors work with the document object which you can access by

typing the keyword document.

This returns the webpage stored in the browser's memory known as

the document object model or DOM.

Alternatively, I can run a council log on the document keyword

to get the same result.

Let's start locating specific elements inside the document object using the query

selector method.

For example, I know this page contains two HTML paragraph elements,

to access the first one I typed document.querySelector.

Then in single quotes inside of the parentheses, I typed the letter P.

When I press enter to run the command, it returns the first P element.

I can then expand this to view the paragraph text contained within,

this method can be used with other html elements, such as the anchor tag.

If I run the same command with the letter a,

I get back the first anchor element on the page.

There is a very similarly named JavaScript selector that allows me to get all

the matches from a web page.

It's the query selector all method,

to demonstrate this, I type document.querySelectorAll.

Inside the parentheses, I passed the p letter as a string and

indeed I get back a result that shows that there are two paragraph tags on this page.

Another useful JavaScript selector is get element by ID which can find

objects in the dawn that match a specified html ID attributes.

For example, let's say we want to find something on this page with the idea of

heading, I typed document.getElementByID.

And then the string heading in parentheses, I press enter and notice that

an H1, element is returned assigned with the html ID attribute value of heading.

Next, let's use a similar method that returns an element based on a specified

class name rather than ID.

We can do this by calling document.getElementsByClassName.

I typed document.getElementsByClassName and then the string txt and parentheses.

Again, I press enter and notice that two results are returned.

The two paragraph tags containing the class name of txt.

If you are a beginner on your journey with JavaScript, it is important to note

that the word element is singular for ID and plural for class name.

Additionally, suppose the element you're looking for

cannot be found in that case you will be returned null for IDs.

An empty collection represented by square brackets for class names.

In this video you learned how to use JavaScript selectors to speed up

the process of finding specific objects in a document.

# Event handling

Let's say you are using a webpage on

your computer and you click a button,

or you tap the like icon

on a friend's picture on your phone.

In JavaScript, the button click and

the like icon tap are examples of user triggered events.

Events are happening all the time.

You can use JavaScript code to listen for these events.

You can even choose the parts of

the page on which you are listening for events.

Let me describe an example.

You might want to listen for

a click event on an Add to Cart button.

Once you capture such an event,

you might want to run some JavaScript code.

For example, I can add

a circle with the number one on it next to

the shopping cart icon to indicate

that there is now one item in the shopping cart.

If the same event gets triggered or fired again,

our event handling code then handles the event by

updating the count in

the circle next to the shopping cart icon.

The circle then displays the number two to

indicate that there are two items in the cart.

In JavaScript, the function that handles

captured events is known as the event handler.

Let me demonstrate how to listen

foreign event by using the add event listener method.

Okay, so I'm back in

the console here and let me demonstrate

two ways to set up an event listener

for HTML elements on a webpage.

One of the easiest ways to listen for

an event is to use the add event listener method.

You can do that on a given HTML element.

For example, suppose I want to listen to

the click event on the body of this example website.

First, I need to get a reference to

the body element where we want to listen to this event.

In the console of the browser developer tools,

I type document.querySelector,

and then the name of the element

inside of the parenthesis.

For this example,

its body, and I enclose it with single quotes.

Next, I want to assign the body object to a variable.

I type const target

equals to assign the body object to the target variable.

I've named this variable target

because it's the target of our click event.

Now I can create a function that

would be run when this target is clicked.

I type function handle click,

and then in the function body,

I console log the string, clicked the body.

The next step is to run

the addEventListener method on the target element.

I type target.add EventListener.

Then inside the parenthesis,

I pass it two arguments.

The first is the event type click as a string value,

and the second is my handle click function.

Okay, so my code is now ready.

Let me test it by clicking the webpage.

Success. Notice that the text is output to the console.

In addition to the addEventListener method,

an alternative way to listen for events is

by using HTML event attributes.

Before doing that, let me first write

a second click handler function named handleClick2.

Once again, I console log

a string inside the function body.

This time it will output the phrase clicked the heading.

Next, I need to open the elements panel of

the developer tools and expand

the view for the div element,

then the heading element should be visible.

I right-click the H1 element and click edit as HTML.

Then after the H1,

I create the attribute by typing

onclick equals followed by

my function name handleClick2

with a pair of parenthesis at the end.

Finally, I click outside the element to

save the change and return to the Console tab.

Notice that when I now

click on the heading in the webpage,

the phrase clicked the heading and

clicked the body is output to the console.

Basically, this single click has

triggered two different event listeners.

But notice that if I click outside of the heading,

it only triggers the event listener that

listens for clicks on the entire body element.

In this video, you learned

how to write an event listener using

the addEventListener method as well as by writing

a function directly as

a HTML event attribute. Great work.

**Exercise: Web page content update**

In this reading, you will learn how to capture user input and process it. You'll be introduced to a simple example that demonstrates how to manipulate information displayed based on user input.

To capture user input, you can use the built-in *prompt()* method, like this:

let answer = prompt('What is your name?');

Once you have the user-provided input inside the *answer* variable, you can manipulate it any way you need to.

For example, you can output the typed-in information on the screen, as an *<h1>* HTML element.

Here's how you'd do that:

let answer = prompt('What is your name?');

if (typeof(answer) === 'string') {

    var h1 = document.createElement('h1')

    h1.innerText = answer;

    document.body.innerText = '';

    document.body.appendChild(h1);

}

This is probably the quickest and easiest way to capture user input on a website, but doing it this way is not the most efficient approach, especially in more complex scenarios.

This is where HTML forms come in.

You can code a script which will take an input from an HTML form and display the text that a user types in on the screen.

Here's how this is done.

You'll begin by coding out a "test solution" to the task at hand:

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

So, you're essentially doing the same thing that you did before, only this time you're also dynamically adding the *input* element, and you're setting its HTML *type* attribute to *text*. That way, when you start typing into it, the letters will be showing in the *h1* element above.

However, you're not there quite yet. At this point, the code above, when run on a live website, will add the *h1* element with the text "Type into the input to make this text change", and an empty input form field under it.

You can try this code out yourself, for example, by pointing your browser to the *example.com* website, and running the above code in the console.

**Remember you can access the console from the developer tools in your browser.**

Another opinionated thing that you did in the code above is: setting my variables using the *var* keyword.

Although it's better to use either *let* or *const*, you're just running a quick experiment on a live website, and you want to use the most lenient variable keyword, the one which will not complain about you having already set the *h1* or the *input* variables.

If you had a complete project with a modern JavaScript tooling setup, you'd be using *let* or *const*, but this is just a quick demo, so using *var* in this case is ok.

The next thing that you need to do is: set up an event listener. The event you're listening for is the *change* event. In this case, the change event will fire after you've typed into the input and pressed the ENTER key.

Here's your updated code:

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    console.log(input.value)

})

This time, when you run the above code on the said *example.com* website, subsequently typing some text into the input field and pressing the enter key, you'll get the value of the typed-in text logged to the console.

Now, the only thing that you still need to do to complete my code is to update the text content of the *h1* element with the value you got from the *input* field.

Here's the complete, updated code:

var h1 = document.createElement('h1')

h1.innerText = "Type into the input to make this text change"

var input = document.createElement('input')

input.setAttribute('type', 'text')

document.body.innerText = '';

document.body.appendChild(h1);

document.body.appendChild(input);

input.addEventListener('change', function() {

    h1.innerText = input.value

})

After this update, whatever you type into the input, after pressing the ENTER key, will be shown as the text inside the *h1* element.

Although this completes this lesson item, it's important to note that combining DOM manipulation and event handling allows for some truly remarkable interactive websites.

# Exercise: Capture Data

var h1 = document.querySelector('h1')

var arr = ['Example Domain', 'First Click', 'Second Click', 'Third Click']

function handleClicks() {

    switch(h1.innerText) {

        case arr[0]:

            h1.innerText = arr[1]

            break

        case arr[1]:

            h1.innerText = arr[2]

            break

        case arr[2]:

            h1.innerText = arr[3]

            break

        default:

            h1.innerText = arr[0]

  }

}

h1.addEventListener('click', handleClicks)

# Moving data around on the web

The modern web consists of millions and millions of web pages, connected services and databases.

There are websites communicating with other websites, getting data from data feeds and data providers, both paid and free.

All of these data flows must be facilitated with some kind of data format.

Around 2001, Douglas Crockford came up with a data interchange format based on JavaScript objects. The name given to this format was JSON, which is JavaScript Object Notation.

Before JSON, the most common data interchange file format was XML (Extensible Markup Language). However, due to XML's syntax, it required more characters to describe the data that was sent. Also, since it was a specific stand-alone language, it wasn't as easily inter-operable with JavaScript.

Thus, the two major reasons for the JSON format becoming the dominant data interchange format that it is today is two-fold:

* First, it's very lightweight, with syntax very similar to "a stringified JavaScript object". You'll learn more about the specifics of this later.
* Second, it's easier to handle in JavaScript code, since, JSON, after all, is just JavaScript.

It is often said that JSON is a subset of JavaScript, meaning it adheres to syntax rules of the JavaScript language, but it's even more strict in how proper JSON code should be formatted. In other words, all JSON code is JavaScript, but not all JavaScript code is JSON.

Besides being a data interchange format, JSON is also a file format. It's not uncommon to access some third-party data from a third-party website into our own code in the form of a json file.

For example, if you had a website with the data on stock price movements, you might want to get the data of the current stock prices from a data vendor. They might offer their data service by giving you access to the file named, say stockPrices.json, that you could access from their servers.

Once you'd downloaded that stringified JSON data into your own code, you could then convert that data to a plain JavaScript object.

That would mean that you could use your web application's code to "dig into" the third-party data-converted-to-a-JavaScript-object, so as to obtain specific information based on a given set of criteria.

For example, if the stringified JSON data was converted to an object that had the following structure:

const currencyInfo = {

    [

        USD: {

            // ...

        },

        GBP: {

            // ...

        },

        EUR: {

            // ...

        }

    ]

}

 You could then access only the data on the USD property, if that's what was needed by you app at a given point in time.

Hopefully, with this explanation, you understand, at a high level, how and why you might want to use JSON in your own code.

It's all about getting stringified JSON data from a server, converting ("parsing") that data into JS objects in your own code, working with the converted data in your own code, and perhaps even stringifying the result into JSON, so that this data is then ready to, for example, be sent back to a server after your code has processed it locally.

## JSON is just a string - but there are rules that it must follow

JSON is a string, but it must be a properly-formatted string. In other words, it must adhere to some rules.

If a JSON string is not properly formatted, JavaScript would not be able to parse it into a JavaScript object.

JSON can work with some primitives and some complex data types, as described below.

Only a subset of values in JavaScript can be properly stringified to JSON and parsed from a JavaScript object into a JSON string.

These values include:

* primitive values: strings, numbers, bolleans, null
* complex values: objects and arrays (no functions!)
* Objects have double-quoted strings for all keys
* Properties are comma-delimited both in JSON objects and in JSON arrays, just like in regular JavaScript code
* String properties must be surrounded in double quotes. For example:

"fruits",

"vegetables"

* Number properties are represented using the regular JavaScript number syntax; e.g

5,

10,

1.2

* Boolean properties are represented using the regular JavaScript boolean syntax, that is:

true

and

false

* Null as a property is the same as in regular JavaScript; it's just a

null

You can use object literals and array literals, as long as you follow the above rules

What happens if you try to stringify a data type which is not accepted in JSON syntax?

For example, what if you try to stringify a function? **The operation will silently fail**.

If you try to stringify some other data types, such as a BigInt number, say 123n, you'd get the following error: Uncaught TypeError: Do not know how to serialize a BigInt.

### Some examples of JSON strings

Finally, here is an example of a stringified JSON object, with a single key-value pair:

'{"color":"red"}'

Here's a bit more complex JSON object:

'{"color":"red", "nestedObject": { "color": "blue" } }'

The above JSON object encodes two properties:

* "color":"red"
* "nestedObject": { "color": "blue" }

It's also possible to have a JSON string encoding just an array:

'["one", "two", "three"]'

The above JSON string encodes an array holding three items, three values of the string data type. Obviously, just like objects, arrays can nest other simple or complex data structures.

For example:

'[{ "color": "blue" }, {"color: "red"}]'

In the above example, the JSON string encodes an array which holds two objects, where each object consists of a single key-value pair, where both values are strings.

# JavaScript Object Notation - JSON

In this video will help you learn to convert a JSON string to a regular

JavaScript object and vice versa.

If you type a piece of JSON code into your browser's console,

it should not produce any errors.

For example, a JSON string with one property named greeting that contains

the value of the string hello.

If I run this code, I get back the same result.

It's worth remembering that JSON is basically just an object represented in

the form of a string.

It has some specific formatting but it's a string nonetheless.

So to work with JSON, it is common to convert it back to a JavaScript object to

work with its properties and methods.

To do this you need to use the global built in JSON object and

its parse method, let me demonstrate this now.

First I will sign the JSON string to a variable

named JSONstr using the const keyword.

Next I'll run the JSON parse method on this variable by typing

JSON.parse and then JSONstr inside the parentheses.

I press enter and notice that a regular JavaScript object is returned.

I want to save this object as a variable.

So I assigned the JSON parse method to a variable named aPlainObj

using the const keyword.

Okay, so now that my JSON string is stored in a regular JavaScript object.

I can manipulate it just like any other JavaScript object.

Let me demonstrate this now by reassigning the value of the greeting property,

I type the object name followed by a dot and then the object property equals hi.

Press enter and notice that the word hi is output.

I can confirm this by inspecting the object and

notice that its structure has been updated.

Okay, that's how you convert a JSON string to a JavaScript object.

You can also go the other way and convert a regular object to a JSON string.

Using the string of my method of the JSON object.

Let me demonstrate this now.

For example, let's start by declaring an object named data and

assign it some properties and values.

To run the JSON string of my method on this object,

I'll type JSON.stringfy and place the objects name inside the parentheses.

The result is a JSON string.

Notice that both the objects keys and

its values are double quoted strings in the JSON syntax.

It's important to remember that while plain JavaScript objects can hold

functions, JSON strings cannot.

Another limitation is that valid JSON doesn't allow the use of JavaScript

comments.

Also when you stringfy a JavaScript object containing a method,

that method will be excluded from the stringfy operation.

In this video you have learned how to convert a JSON string to a JavaScript

object with the parse method.

And how to convert a JavaScript object to a JSON string using the stringfy method.

If you work with retrieving data from APIs converting the JSON strings to JavaScript

objects will be a standard workflow.

You can then easily access the objects properties programmatically this is

a vital tool in your tool belt, and I encourage you to practice this.

**Convert JSON to JavaScript object using JSON.parse() and put the JSON in the parenthesis.**

**Concert JavaScript object to JSON using JSON.stringify() and put the JavaScript object in the paranthesis.**

# Module summary: Programming Paradigms

You've reached the end of this module on

programming paradigms. Great work.

During this module, you explored the core features of

functional programming and object-oriented programming.

Let's review the module by

summarizing the major items you encountered.

The various styles of

computer languages are known as programming paradigms.

Having studied programming paradigms in this module,

you should be able to describe

functional programming and how it

can be used to solve a problem by

separating data from functions.

Having explored functional programming,

you moved on to another popular paradigm.

Object-oriented programming, often referred to as OOP.

You discovered how OOP

differs from functional programming.

You studied object-oriented programming principles.

In programming, scope determines which parts of

the code are accessible and which parts are inaccessible.

You learned how the scope of variables changes when

you use the keyword Var in ES5 JavaScript,

and when you use the keywords,

Let and Const in ES6 JavaScript.

You should now be familiar with how

the scope chain works in JavaScript.

You should also now be able to

identify how some of the different types of scope work,

including global and local.

Much of the development work in

JavaScript involves building objects.

Classes are essentially a blueprint that you

can repeatedly use to build new objects.

You can also pass parameters to

the class methods and then use

them the same as with regular functions.

Constructors tell JavaScript how

you want your objects to be built.

You should now be able to utilize

constructors on built-in JavaScript objects.

You learned that inheritance exists in JavaScript,

and the inheritance model revolves around the prototype.

Additionally, you explored how

JavaScript code uses modern features like spread,

rest, template strings, and modules.

You build code that manipulates

the DOM and handles events.

Finally, you used JSON in JavaScript.

That was a lot to cover,

but you should now be more familiar with

programming paradigms in JavaScript. Great work.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN: Modules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

[Nodejs.org official docs on CommonJS](https://nodejs.org/api/modules.html#modules-commonjs-modules)

[MDN: DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

[MDN: Document.querySelector](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector)

[MDN: Event](https://developer.mozilla.org/en-US/docs/Web/API/Event)

[MDN: EventTarget.addEventListener](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)

[MDN: Working with JSON](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON)

# Other JavaScript environments - node & NPM

In 1886, the first internal combustion engine-powered car

was invented by Karl Benz.

Soon, there were many cars

that used the combustion engine,

airplanes, motorbikes, and boats followed shortly after.

People's ingenuity led to using

the combustion engine in

places that the inventor might not have imagined.

The same happened to JavaScript.

New JavaScript environments such as

Node and npm are now available.

JavaScript's home is in the browser.

That's the environment in which it

was used for several years.

Essentially, it was a front end only language.

Then in 2009, a developer named Ryan Dao

decided to use Google's JavaScript V8 engine

and make it work on the server.

It's not always easy being an innovator.

Many doubted whether it was even possible.

However, the idea caught

on and more people started getting on board.

This is how Node.js was born and how

JavaScript became a language

for both front-end and back-end.

Node.js is a separate standalone environment.

This means that Node.js can run in multiple settings.

For example, on the command line,

in a desktop application,

or on the back end of a web app.

Before the introduction of Node.js,

developers had to build backends in

other technologies and languages such as PHP,

Python, C-sharp, Ruby, and Java.

After Node.js became available,

it was possible to use JavaScript on

the backend or on the server-side.

This means that today you can

write full-stack JavaScript programs.

In other words, you can write

JavaScript on the client and on the server.

Node.js comes with a package manager called npm,

which stands for Node Package Manager.

The package manager allows you to use a large number of

libraries and frameworks as Node.js modules.

An npm module is a standalone piece of

code that has been published on the npm website.

Sometimes an npm module is

also referred to as an npm package.

Now that you've learned about Node.js,

you may be wondering how you can use it locally.

Node.js and npm are either pre-installed on your machine,

or you need to install them.

Once installed, you can interact with

Node.js and npm from the command line.

For example, you can run

the node command inside your computer's command line.

This is also called a shell, or a terminal.

In the same way, you can run the npm command.

You may be wondering why you need to run those commands.

Well, this is because you use

the node command to run a JavaScript file,

or to directly execute JavaScript code.

You use the npm command to install

any node module from the npm repository.

For example, you could install

the module named lowercase.

Once installed, you can import this module and use

its functions to transform

strings to lowercase in your code.

When you want to start a new project, first,

open a folder on

your machine where you want to place your project files,

then run the npm command.

These projects can be different shapes and sizes,

but they all have at least one thing in common,

the package.json file that gets

created after you run the npm command.

The package.json file holds all the instructions on

all the node modules that are pulled from

the npm repository of open source modules.

There are about 11 million modules in the npm repository.

It means that you can get thousands of hours worth of

other developers' coding by running

the npm install command and adding the package name.

Examples of libraries you can install include React,

Webpack, Bootstrap, and Angular Core.

The package.json file updates

when you install a new package.

It keeps track of everything you need

to have installed in your project.

This makes such projects easily portable.

For example, if you have built

a project with a specific number

of different node packages,

they're all listed inside the package.json file.

All you need to do is share this file with,

for example, your co-workers.

They can have the exact same setup on their machines

simply by running the command npm install.

This install command reads the contents of the

package.json and installs all the necessary packages,

also referred to as dependencies.

Sometimes dependencies also come

with their own dependencies.

It often happens that when you run

the npm install command,

several 100 megabytes worth of node packages get

installed into your project

under the node modules folder.

In this video, you learned about the origins of Node.js.

You should also be able to describe what Node.js and npm

are and how to use

the basic Node and npm commands on the command line.

# What is testing?

When you're building something,

you want to make sure it works,

and you make sure that something works by testing it.

It's the same with JavaScript code.

You need to make sure that

your JavaScript code works before you launch it.

In this video, I'll show you how

web developers test their JavaScript code.

By the end of this video,

you'll be able to: identify

the various reasons why

a developer would want to test their code,

explain how the red green refactor cycle

relates to testing,

differentiate between manual and automated testing,

and demonstrate an understanding of unit testing.

How do web developers test their code?

Let's find out by coding a custom concatStrings function.

For example, I can create

a concatStrings function to

join any two strings that I give it.

In this case, I want the function to join string

A and string B that are received as arguments,

and return the result.

For example, I could write concatStrings a, b, c,

and then d, e, f,

and the returned result would be abcdef.

When the function receives number 1 and 2,

it returns a three as a number instead of

a string of one and two together i.e, 1, 2.

I can also signal what I want

my function to do by adding comments to it.

Adding comments will help my team understand and

remember the expected behavior

that I had for the function.

A more detailed description

would be a good way to make it

more obvious as to what

my intention is for each argument.

In the comments, I can even specify

the expected behavior when

specific values are given to the function.

Adding comments is a step in the right direction,

but it has downsides.

It allows me to write anything I like,

so there are no limits on ambiguity.

Also, the common doesn't need

to follow any set structure.

There are many custom testing frameworks

available in JavaScript.

One of the strengths of such frameworks is that I don't

have to use comments to describe my expectations.

The test syntax itself becomes expectation documenting.

When I write tests,

those tests are a better alternative to comments in

my source code because they specify what

expectations my source code is trying to satisfy.

Tests are also callable,

meaning I can execute

tests to check if expectations are met.

As I demonstrated earlier,

I could execute the concatStrings function

with the first argument being the string abc,

and the second argument, the string def.

I would then expect the function to

return abcdef, when I call it.

To write this as an expectation

in some testing frameworks such as JEST,

I can use the function that has the name expect.

I then pass the call to

the concatStrings function with the specific arguments.

Then I add that to the function,

which is another testing function,

and I pass it the value of what I'm

expecting this code to produce.

I'm essentially stating that I expect

the calling concatStrings with abc and def,

will return the value abcdef.

Testing in JavaScript lets me verify

that the function is behaving in the way I intended.

Testing code in this way ensures three things.

Conciseness as it's straightforward and to the point,

since there are only two function calls to

explain what is the expected result.

Clarity, because you know

exactly what arguments you're providing,

and repeatability, as you can run it

again and again with the same arguments each time.

Now, I could run

multiple function calls using this syntax.

For example, I could have expected concatStrings

123 and 456 as arguments,

and then the function to be with the argument 123456,

and in each instance,

the expectations will be correct,

and the code will behave as expected.

In testing terminology, you would

say that your tests are passing.

But I'm only halfway there because it's still

possible from my code to fail an expectation.

For example, if I run the expect method and pass it

an invocation of the concatStrings function

with numbers 1 and 2 as arguments,

my expectation of 12 being the results will fail.

This is because when I use

the plus operator with two values of the number type,

it performs the mathematical operation of addition,

instead of joining the two numbers

together to form the number 12,

like it would form abcdef.

If I gave it the arguments of abc and def.

When tests fail, you say that they are red,

and when they pass, you say that they are green.

If a test fails,

then it's a sign that I need to write the code

in such a way that it passes its test.

Once my test passes,

I need to improve both the app's code and the test code,

but without changing the behavior of either.

This is known as refactoring.

Refactoring is the process in which I

write my feature code so that it runs more

efficiently or that it's easier to read and thus

easier to understand for other programmers on the team.

This is done without affecting

the results that the code produces.

The only obvious change should be perhaps the test itself

taking less time to run

because I've optimized my source code.

You want to always write your code in

a way that meets all expectations.

It's most likely that some of your tests

will be red and some of them will be green.

Red tests are a guide as to how you need to

improve your code to cater for unmet expectations.

As you continue to refine

your code in response to red tests,

it becomes a cyclical activity.

This is often described as the red-green-refactor cycle.

This cycle is the basis of

the test-driven development or

TDD approach to programming.

Let me explain the TDD approach.

First, you write a failing test,

then you write your source code so

that the previously failing test now passes.

Finally, you optimize

your source code without changing its results.

There are many advantages of having

code that tests other code.

For example, you can run it as many times as you want.

You can run the testing code automatically.

The tests can be repeated without

significant cost in time or effort.

To sum up, testing is how you verify

the expectations you have

regarding the behavior of your code.

# Types of testing

As a developer, you are tasked to test a piece of software,

how would you do it in the next few minutes.

I am going to explain what software testing is.

I'm going to introduce you to the three most used types of testing.

You might say that testing a piece of software at a high level means making sure

that it behaves as expected in any modern software development project.

These expectations are recorded as software requirements.

There are many aspects of software testing, but for now let's think of it as

the act of confirming that it works as outlined in the software's requirements.

Now that I have a working definition of testing,

let's think about how you might approach it for a UX designer.

Testing might mean making sure that the website looks and behaves as expected.

For project manager on a software project testing might mean that a specific piece

of software works well with other parts of your system for a software engineer.

Testing might mean writing code that doesn't break the existing

functionality is bug free and satisfies the requirements as set out in

a given task based on what your motivations are.

There are several ways that you can test your software projects practically.

This means that there are different kinds of testing.

Now, I'll discuss the following three types of testing into in testing R.

E to a integration testing and unit testing.

Let's start with into in testing a real life example of E two EE testing

would be if a laptop manufacturer, let his employees open some off

the assembly line laptops turn them on and use them just like a normal user

would do to make sure that the entire in product behaves as it should.

More specific to web development into.

In testing tries to imitate how a user might interact with your app.

This means that in Italy testing you need to open your web application in

a browser and then test it by interacting with the page the same

way a user might interact with it.

For example, clicking on the log in button are going through the process of

adding an item to the shopping cart.

In other words, you're testing the entire

finished software product from the perspective of the end user.

Now here's something you may find interesting.

The person testing the app doesn't have to be a developer.

Finally, E two E tests are the slowest and take the most time to set up and run.

Here are a few examples of eat away testing frameworks available on

the market.

Web driver Js Protractor and Cyprus.

Next I'll discuss integration testing.

Integration testing is testing how parts of your system interact with other

parts of your system.

In other words, it's testing how separate parts of your apps work together.

Let's explore two examples of integration testing software react testing,

library and enzyme integration tests are faster and

cheaper than E two E tests but not as fast or as cheap as unit testing.

What is unit testing.

Let's have a look.

Unit testing is the process of testing the smallest units of your

source code in isolation.

A good example of this is functions.

A unit is the smallest piece of code that you can test separately

from the rest of the app.

Practically the smallest unit of testable code in Js is usually a function or

a method.

Unit tests are self contained.

They're meant to test code in isolation,

preferably separate from the rest of your app.

This makes unit tests fast to run and easy to write.

So That was a brief overview of the three different kinds of tests.

Together, they are sometimes represented as a three level testing pyramid.

First at the base of the Pyramid are the unit tests which have the highest

speed and the lowest cost.

Then in the center of the pyramid are the integration tests of average speed and

expense.

And finally at the top of the pyramid are the E two E tests,

the slowest and most expensive tests to run.

You are now familiar with the three most common ways that you can test your applications.

# Introduction to Jest

At this stage, you might be familiar with the concept of

testing your code to

make sure it's working as you intend it.

You might also have come across

some syntax of testing frameworks.

But what about a language like

JavaScript that has no built-in testing functionality?

For JavaScript, you can use testing frameworks like Jest.

Over the next few minutes,

I am going to introduce you to

the features of the Jest framework,

code coverage, mocking, and snapshot testing.

It is quite a mouthful. Let's get started.

JavaScript doesn't have built-in objects or

methods that would allow for tests to be written.

Therefore, many different libraries

have been built to tackle the issue of testing.

Some examples of these libraries include Jasmine,

Mocha, Karma, and qUnit.

Then there is the one that you will cover now,

namely the Jest testing framework.

Jest is a JavaScript testing framework.

It's often used for testing code like React,

a JavaScript library maintained by Meta and

a community of individual developers and companies.

Besides plain JavaScript and

React code just allows you to test Babel,

TypeScript, Node, Angular,

Vue, and various other frameworks.

Jest also supports code coverage.

Code coverage is a measure of

what percentage of my code is covered by tests.

If I say that I have an 80 percent code coverage,

that means that only one-fifth of

my entire code base is not covered by tests.

But even 100 percent code coverage doesn't

mean that you have tested

for every conceivable expectation.

It just means that there are

some expectations tested for each line of my code.

Still, code coverage is a handy tool to

gauge the amount of

my code base that's included in tests.

The higher the code coverage,

the lower the chance of having unidentified bugs.

As a rule, the higher the percentage of code coverage,

the lower the amount of time required to write new tests.

This, however, depends on whether

there are incomplete software requirements

pending or if you are going to

receive more requirements in the future.

Next, let's cover the concept of mocking.

Mocking allows you to separate the code

that you are testing from it's related dependencies.

In other words, you can use the mocking features

to make sure that your unit testing is stand-alone.

For example, you can test

the front end functionality of your web app by

mocking the data as if it came back from

a server when in fact it came from the client.

Mocking is especially helpful because very

often web applications are built by teams of developers.

Some developers work on the backend of

a feature and others work on the front end.

This could result in bottlenecks.

Take an example where the team

decides to build a new feature that

lists the address book of

users of the app on the front end.

The actual user related data

for this feature would come from the server.

But what if a back-end developer was a

bit late in developing their part of the feature?

Then a front end developer would be stuck waiting for

the back-end developer to complete

their work before the front-end code can be built.

With mocking you can avoid this bottleneck.

Mocks, allow you to

pretend that the users are already there.

The needed data comes from the

mock rather than from the backend.

This allows the front-end developers to

finish their site of the new feature independently.

In certain cases, developers can

use mocking to ship features faster.

Some libraries, such as sign-on,

focus specifically on mocking.

But the great thing about Jest is that you use it's

mock functions without any additional installations.

In Jest you use mocking by employing Jest mock functions.

It's also easy to test asynchronous code in Jest.

There are no difficult setups and tests are

relatively easy to code

even for newcomers to the framework.

Finally, Jest allows you to perform snapshot testing.

Snapshot testing is used

by developers to verify that there are

no regressions in the DOM of our apps

after some changes to the code base are made.

You're now familiar with the concept of testing

your JavaScript code using

the Jest testing framework. Great work.

# Writing tests with Jest

In this video, you'll explore how to install

the packages needed to test

your JavaScript code and the Jest framework,

as well as how to set up a test.

Let's say I need to write a function that

takes a value and adds five to it.

I'll start by creating a new file

and naming it addFive.js.

The code in the addFive function starts with function

addFive and then vowel in parentheses.

In between curly braces,

I have returned vowel plus five on the next line,

and then on a new line,

I have module.exports equals add five.

This is a simple function which will make it

easier to analyze when I write tests for it.

I've also added a line to export this function so

that it can be used by other files in this project.

Now I'll switch over to using

the jest testing framework to write

some expectations of how this function should behave.

Before doing so, I'll check if I have

no JS and NPM installed on my system.

In the terminal tab,

I type node -,

- version to check for NodeJS.

And it shows that I have version 16.15.

Now I check for NPM and it tells

me I have version 8.5 installed.

One more step before I continue is to check if

Jest installed by typing Jest - - version.

In this case, it returns command jest not found,

meaning that it is not globally installed on my machine.

So I'll install it for this project using only NPM,

since jest needs NodeJS to run and

NPM as a way to install

Node modules into my web projects.

I'll first type and run NPM -

y to add a package.json file.

The - y File automatically answers yes to

all the questions that init

command asks during installation.

Now I have my package.json file.

In other words, I have a way to keep track

of node modules that this project depends on.

Since I want to use the jest testing library,

I'll need to install it locally,

which means installing it for this project.

So I run NPM install -,

-, save, -, - Jest.

After a few moments,

the package.json file will be

updated with the addition of the jest testing library.

I also have a folder called Node underscore modules,

which is where the code for

all the modules in this project is stored.

In the package.json file,

I'll need to make a few changes to the script section.

In the test entry, I replace the text that is assigned

to test with jest within double-quotes and save it.

Now when I run the command NPM run test,

it will run the Jest command in this folder.

What does this Jest command do?

It runs tests for the code in my project.

Right now, I only have a single file, addFive.js.

So it makes sense to create

a test file for this one JavaScript file.

I'll name it addFive.test.js.

In the naming convention I use,

I add the dot test just before

the dot js section of the file's name.

To set this file up,

I'll first import

the exported addFive function by typing const,

addFive equals require, and then in parentheses,

backtick dot, forward slash, addFive, backtick.

The reason for the forward-slash is to indicate that

the function is in the same folder as a test file,

and I've omitted the dot js

because Node understands what I mean,

even if I don't use it.

So my expectation is that this function

returns whatever value I input plus five.

To check, I'll use the test method with

a string as a parameter that describes the test.

This string will be output in the command line when I

run the test along with the words pass or fail,

and the function to run when I

execute the NPM run test command.

The function will check if my expectation is correct.

The string will be output in the command line when I

run the test along with the words pass or fail,

and the function to run when I

execute the NPM run test command.

The function will check if my expectation is correct.

I have test and as the first argument,

I have the string returns the number plus five,

which is what I want to output on

the command line when the test is run.

Then as a second argument,

I have a function which checks my expectation,

I'll save and run the test and I get back a pass result.

In this video, you learned how to ready your system

for testing in Jest and how to set up and run a test.

# Lab Instructions: Unit Testing

### ****Tips: Before you Begin****

#### **To view your code and instructions side-by-side**, select the following in your VSCode toolbar:

* View -> Editor Layout -> Two Columns
* To view this file in Preview mode, right click on this [README.md](http://README.md) file and Open Preview
* Select your code file in the code tree, which will open it up in a new VSCode tab.
* Drag your assessment code files over to the second column.
* Great work! You can now see instructions and code at the same time.
* Questions about using VSCode? Please see our support resources here:  
  [Visual Studio Code on Coursera](https://www.coursera.org/learn/programming-with-javascript/supplement/roMvE/visual-studio-code-on-coursera)

#### **To run your JavaScript code**

* Select your JavaScript file
* Select the "Run Code" button in the upper right hand toolbar of VSCode.  
  Ex: It looks like a triangular "Play" button.

## Task 1: Add Jest as a devDependency

Open terminal. Make sure that it's pointing to jest-testing directory.  
Install the jest npm package using the npm install command and the --save-dev flag.  
Verify that the installation was completed successfully by opening the package.json file and confirming that the "devDependencies" entry lists jest similar to the following:

"devDependencies": {

"jest": "^28.0.0"

}

## Task 2: Update the test entry

In the package.json file, locate the "scripts" entry, and inside of it, update the test entry to jest.

## Task 3: Code the timesTwo function

Open the timesTwo.js file and add a function named timesTwo. The function should take number as input and return the value 2 multiplied by the number. Export the timesTwo function as a module.

## Task 4: Write the first test

Code a test call with the following arguments:

1. The description that reads: "returns the number times 2".
2. The second argument should expect the call to the timesTwo function, when passed the number 10, to be 20.

## Task 5: Run the first test

With the terminal pointed at the jest-testing directory, run the test script using npm.

**package.json file**

{

  "name": "jest-testing",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "jest"

  },

  "author": "",

  "license": "ISC",

  "devDependencies": {

    "jest": "^29.0.1"

  }

}

**timesTwo.js file**

// Task 1: Code the timesTwo function declaration

function timesTwo(number) {

    return number \* 2;

}

// Task 2: Export the timesTwo function as a module

module.exports = timesTwo;

**timesTwo.test.js file**

const timesTwo = require('./timesTwo');

// Write the first test

test('returns the number times 2', () => {

    expect(timesTwo(10)).toBe(20);

});

Run the test by running **npm run test** from terminal.

# TDD (Test-Driven Development)

Every piece of software is built according to formal or informal requirements.

The purpose of the requirements is to explain,

in human language the intricacies of what the piece of software does.

So how did the requirements tie in with the practice of test driven development?

Let me explain test driven development.

R T D D for short is a streamlined process of writing code that will

satisfy some requirements.

Let's unpack this process a bit on a high level.

A software development teams work consists of the following receiving requirements

which will become a feature of the app that's being developed.

Writing a failing test for that to build feature before it gets built.

Making this failing test pass by coding that given feature in comparison

with the traditional development process,

that TDD approach might seem somewhat upside down.

I am now going to demonstrate the TDD approach by writing a failing test for

a javascript file and

then writing code to make this test pass to understand how TDD works.

Consider the following real life situation.

Suppose you have to perform a task, drive your car to work.

You leave your house and walk up to your car,

only to find out that you don't have your car keys with you.

Then you remember you left your car keys in the cabinet and

you simply forgot to take them what you did there.

In this imagined scenario is the opposite of TDD.

You first walk to your car and only then did you check if you had your car

keys if you did these things using the TDD approach,

you would do the following first.

You check or test if you have your keys with you.

Your test fails because you don't have them.

They're in the cabinet.

Then you perform the action of getting your keys from the cabinet.

Finally, you check or test if you have your keys this time you have them.

So your test now passes.

What is described here is the essence of TDD.

Let's go through these steps again this time,

pretend that you're implementing TDD in your code.

Imagine that you need to write code in a test driven way.

Since your coding the TDD way you first write the test,

even before you've written any actual implementation, for

example, you test if a function named status of keys exists,

you then use some testing functions from a testing framework.

Since you haven't written your source code implementation.

The test fails.

Next you run your test.

The test fails because there's no status of keys.

Function declared.

The logic of your test code is expect that the function status of

keys exists in your source code.

You declare a function named status of keys.

You run the test.

Again, it checks if there is such a function and it confirms it exists.

So the test passes, it's important to note that one of the rules

of TDD is that you should write as little code as possible to make the test pass for

this test to pass, it's enough to just declare a function with the name next you

receive another requirement which is as follows except a keys variable,

which should be set to true and console log the keys variable.

So the requirement states the status of keys function should accept

a previously declared keys variable, which should be set to true.

The status of keys should then console log the value of the keys variable.

So you write another test which fails again since you have yet to write

the implementation, you write it and your previously failing test now passes.

Finally, you examine your function code and

realize that the indentation is all wrong.

There are also too many unnecessary comments.

So you clean up your code and

run the test again to confirm that you haven't accidentally made any errors.

The test still passes, so everything is okay.

That is the TDD approach in a nutshell, let's go over it one more time.

In a scenario as a member of the development team,

your task is to read the requirements for the software that you are writing.

The requirements are passed to you by the project manager.

So you get your first task for the day and start coding the TDD way.

First you read the new requirement.

Next you write a failing test, then you update your source code.

So it resolves the requirement After that you run a test that passes Finally,

you re factor your implementation.

This process is usually explained in three words.

Red, Green.

Re factor, red represents the failing test.

Green on the other hand,

represents the passing test after you make updates to the source code.

The re factor represents the final tweaks to the code that don't change

implementation details, which can always be confirmed

by running another subsequent test when implemented correctly.

TDD brings huge benefits to an organization because it allows for

automated testing in any platform projects grow bigger over time and become complex.

Making sure that all the tests are passing is a strong signal that the current

requirement and all the previous requirements for

this piece of your app have been delivered successfully and that nothing is breaking.

Test driven development has many advantages.

Here are a few with TDD, you are minimizing regressions that is accidental

bugs introduced to old code by coding a new requirement and you also have

proof that your new implementation is not breaking other parts of the app.

You can automate these tests easily and thus keep verifying again and

again that the system works as expected.

You can test your implementations with various inputs and the tests become

a specific kind of documentation for the new members of your team.

In this video, you learned how test driven development, r t d d works,

You also covered the steps involved in the TDD approach and

that it is implemented in Red Green re factor cycles

# Module summary: Testing

You've completed the testing and compatibility module.

Well done.

in this module, you got an introduction to code testing.

This was an insight into how web developers test their JavaScript code.

Join this module, you explored the importance of testing and

you examined Javaccript testing methods.

Let's finish up now with the recap of the material included in this module and

summarize the content by focusing on the key learning points.

The history of the development of the JavaScript language was outlined and

you learned that today it is possible to write full stack programs with JavaScript.

You should now know how to describe what no J S and

N PNR and install and set up node JsSand NPM.

In any development environment you need to check that your code works

properly before you move to the next phase of its development or

release a completed program to users.

An example of using the contact strings function to join two strings served to

demonstrate how commenting can help to indicate your codes intended behavior and

how testing in JavaScript verifies that a function is behaving in the way that it

was intended.

Having completed that section of the module, you should now be able to identify

reasons why a developer would want to test their code,.

Differentiate between manual and automated testing, identify and

describe the three different kinds of testing and

explore the features available to web developers within the jest framework.

Additionally, you should be able to understand the concepts of mocking and

snapshot testing, explain how to test JavaScript files with jest and

demonstrate how to write a unit test.

You've put a tremendous effort into getting this far and

gained some critical skills by learning about the need for code testing.

You've also explored various methods used for JavaScript testing.

Great work.

You've completed another key part of your software development education,

keep the learning momentum going and be sure to join me for the next module.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[MDN: Server-side website programming](https://developer.mozilla.org/en-US/docs/Learn/Server-side)

[Nodejs.org docs website](https://nodejs.org/api/documentation.html)

[Jest testing framework website](https://jestjs.io/)

[Cypress testing framework](https://www.cypress.io/)

[npm website](https://www.npmjs.com/)

[Unit testing in JavaScript](https://www.browserstack.com/guide/unit-testing-in-javascript)

# Recap Programming with JavaScript

In this course, you discovered how

to program with JavaScript.

Let's take a few minutes to recap what you learned.

In the opening module,

you discovered real-world applications of

JavaScript and uncovered possible job opportunities.

You also learned how to install

VS code on Windows so that you could code in JS.

You did learned about the importance of JavaScript in

programming and the different libraries

used by JavaScript developers.

As you progress through the lesson,

you encountered key concepts in JavaScript.

By engaging with these concepts you

discovered how to identify and declare variables,

differentiate between data types,

and recognize math and logical operators.

You also learned how to perform basic arithmetic using

the number data type and utilize strings and Booleans.

In the next lesson,

you explored conditionals and loops.

During your exploration of these concepts,

you learned how to script different forms of JavaScript

conditional statements and work

with different types of loops,

such as for,

while, and nested.

In Module 2 you

explored the building blocks of a program.

You began this module with a lesson

on arrays, objects, and functions.

During this lesson you learned how to explain

the purpose of functions and describe their benefits,

outline the process for building

an array and accessing its contents,

and you now understand

the array characteristics and

concatenation operators of strings,

and you can now also build and call functions,

create custom objects,

explain how math objects work,

and interact with arrays and objects.

You then moved onto the next lesson in which

you explored the concept of error prevention.

During this lesson, you discovered how to

recognize the differences between bugs and errors,

demonstrate try-catch blocks,

and explain the concepts of undefined,

null and empty string values.

The next module on programming paradigms

began with the lesson on functional programming,

in which you learned how to

explain how the functional paradigm works,

outline the concept of recursion,

describe how scope operates,

and explain different forms of scope

such as let, const, and var.

In the next lesson,

you explored object oriented programming and

learned the principles of object oriented programming,

how to deploy classes in JavaScript,

build and utilize constructors,

and make use of inheritance.

You then explored advanced JavaScript features

in which you learned how to destruct arrays and objects,

utilized for-of loops and objects,

explain the concepts of

template literals and data structures,

and demonstrate the use of the spread and rest operators.

Finally, you then explored how

JavaScript operates in the browser.

During this lesson, you learned how to

manipulate JavaScript in the DOM,

make use of JavaScript interactivity in the browser,

work with events,

and capture and move data around the web.

In the next module,

you looked at testing and compatibility.

The lesson began with an introduction

to testing in which you learned about

the concept of testing and how

to practice different types of testing,

how to write a unit test,

the process for writing tests with Jest,

and how to apply test-driven development.

You then reviewed JavaScript

testing challenges in which you learned

how to navigate other JS environments like Node and NPM,

the concepts of webpack and transpiring,

the process for working with

arrow functions and classes in React,

and you discovered the role

the DOM plays in testing challenges,

and finally, you discovered how to write tests with Jest.

Well done on completing this recap.

Now it's time to try out what you've

learned in the graded assessment. Good luck.

**# Lab Instructions: Little Lemon Receipt Maker**

> **### \*\*Tips: Before you Begin\*\***

> **#### \*\*To view your code and instructions side-by-side\*\*, select the following in your VSCode toolbar:**

> - View -> Editor Layout -> Two Columns

> - To view this file in Preview mode, right click on this README.md file and `Open Preview`

> - Select your code file in the code tree, which will open it up in a new VSCode tab.

> - Drag your assessment code files over to the second column.

> - Great work! You can now see instructions and code at the same time.

> - Questions about using VSCode? Please see our support resources here:

> [Visual Studio Code on Coursera](https://www.coursera.org/learn/programming-with-javascript/supplement/roMvE/visual-studio-code-on-coursera)

> **#### \*\*To run your JavaScript code\*\***

> - Select your JavaScript file

> - Select the "Run Code" button in the upper right hand toolbar of VSCode.

> Ex: It looks like a triangular "Play" button. <br><br>

<br>

**## Assignment Instructions**

In this exercise, you will work with some data provided as an array of objects, listing information about dishes available in the Little Lemon restaurant.

You will need to write a function declaration which will be able to do two things:

    If the function is called with the argument true, it will output the names of the dishes and calculate their final price (including 20% tax)

    If the function is called with the argument false, it will output the names of the dishes and give their prices without the additional tax

The expected outcome is that all the dishes' names and prices will be shown in the console output.

The text below shows the output that your code should produce:

Prices with 20% tax:

Dish: Italian pasta Price (incl.tax): $ 11.46

Dish: Rice with veggies Price (incl.tax): $ 10.38

Dish: Chicken with potatoes Price (incl.tax): $ 18.66

Dish: Vegetarian Pizza Price (incl.tax): $ 7.74

Prices without tax:

Dish: Italian pasta Price (incl.tax): $ 9.55

Dish: Rice with veggies Price (incl.tax): $ 8.65

Dish: Chicken with potatoes Price (incl.tax): $ 15.55

Dish: Vegetarian Pizza Price (incl.tax): $ 6.45

<br><br>

**\*\*Step 1:\*\*** In the function `getPrices()`, give it the parameter of `taxBoolean`.

**\*\*Step 2:\*\*** Inside the `getPrices()` function, code a for loop that will loop over all the objects inside the `dishData` array.

**\*\*Step 3:\*\*** Inside the for loop, declare a `finalPrice` variable, without assigning it a value.

**\*\*Step 4:\*\*** Still inside the for loop, add an if condition, checking that the `taxBoolean` is set to `true`. Inside the if block, multiply the following: \* the price of the currently looped-over object from the `dishData` array, and \* the tax value. Assign the multiplied value to the finalPrice variable.

**\*\*Step 5:\*\*** Right after the if condition, add an else if, checking if the value of `taxBoolean` is false. Inside this condition's block, assign the currently looped-over dish price property in the `dishData` array to the `finalPrice` variable.

<br><br>

**\*\*Step 6:\*\*** Code the `else` case, and inside of it, add two lines of code:

A console log of the string:

- "You need to pass a boolean to the getPrices call!"

- return (to "jump out" of the further function execution)

<br><br>

<b>Step 7:</b> After all the conditional's statements, but still inside the for loop, code another console log with four arguments:

1. The string `"Dish: "`

2. The value of currently looped-over dish object's name property

3. The string `"Price: $"`

5. The value of the `finalPrice` variable

<br><br>

**\*\*Step 8:\*\*** You're finshed with the `getPrices()` function, and now you're ready to code another function. Give the `getDiscount()` function, two parameters: the `taxBoolean` and the `guests` parameter.

**\*\*Step 9:\*\*** Inside the `getDiscount()` function,  on the very first line of its body, invoke the `getPrices()` function, passing it the `taxBoolean` as an argument.

**\*\*Step 10:\*\*** On another line, you need to implement your defensive coding skills, and check that the type of the `guests` parameter is 'number' and  that the value of the guests variable is greater than zero and less than 30.  If all these conditions return true, code the body of the conditional as described in the next step. If they don't all return true, code the body of the else conditional as instructed in step 12.

**\*\*Step 11:\*\*** Inside the if statment, declare a new variable, named `discount`, and set it to 0. On the next line, add another if...else if: in the first if, you'll check that the value of the `guests` variable is less than 5. If that's the case, reassign the value of the discount variable to 5;

- Inside the else if condition, check that the value of the guests variable is greater than or equal to 5 - if that's the case, reassign the discount variable to 10.

- Console log the following after closing your else-if statement: `'Discount is: $' + discount);`

**\*\*Step 12:\*\*** In the else condition, console log the following string: `'The second argument must be a number between 0 and 30'`. Since you've finished declaring both the `getPrices()` and the `getDiscount()` functions, you can now invoke the `getDiscount()` function several times, with various combinations of arguments, to check the behavior. <br>

Here are two examples:

- `getDiscount(true, 2) `

- `getDiscount(false, 10)`

What happens when you don't pass-in any arguments?

What happens when you pass values that are not expected?

**Finished code**

// Given variables

const dishData = [

    {

        name: "Italian pasta",

        price: 9.55

    },

    {

        name: "Rice with veggies",

        price: 8.65

    },

    {

        name: "Chicken with potatoes",

        price: 15.55

    },

    {

        name: "Vegetarian Pizza",

        price: 6.45

    },

]

const tax = 1.20;

// Implement getPrices()

function getPrices(taxBoolean) {

    for (key of dishData) {

        var finalPrice;

        if (taxBoolean == true) {

            finalPrice = key.price \* tax;

        } else if (taxBoolean == false) {

            finalPrice = key.price;

        } else {

            console.log("You need to pass a boolean to the getPrices call!");

            return;

        }

        console.log("Dish: " + key.name + " Price: $" + finalPrice)

    }

}

// Implement getDiscount()

function getDiscount(taxBoolean, guests) {

    getPrices(taxBoolean);

    var correcttype = typeof (guests) == 'number';

    var guestnumber = guests > 0 && guests < 30;

    if (correcttype == true && guestnumber == true) {

        var discount = 0;

        if (guests < 5) {

            discount = 5;

        } else if (guests >= 5) {

            discount = 10;

        }

        console.log('Discount is: $' + discount);

    } else {

        console.log('The second argument must be a number between 0 and 30');

    }

}

// Call getDiscount()

getDiscount(true, 2);

getDiscount(false, 10);

getDiscount();

getDiscount(false, "10");

# Congratulations on completing the course Programming with JavaScript

Congratulations on completing

the programming with JavaScript course.

You've worked hard to get here,

and developed a lot of new skills along the way.

Let's list some of the key skills you've gained.

You began with an introduction to programming,

and you should now be able to explain

the capabilities and uses of JavaScript,

describe the background and history of JavaScript,

explaining the importance of ECMA and ECMAScript,

and describe JavaScript libraries and their uses.

You did received an introduction to

core concepts of JavaScript coding,

and should now know how to code

single-line and multi-line comments with

semicolon syntax and using variables,

data types, operators,

numbers, Booleans, and strings.

You further extended your coding abilities by

writing statements using conditionals and loops,

using arrays, objects and functions,

and enabling error prevention in your code.

Programming paradigms introduced you to

functional programming and object oriented programming,

advanced JavaScript features and

the operation of JavaScript in a browser.

Finally, with testing and compatibility,

you discovered the purpose of testing,

explored types of testing,

and studied JavaScript testing

challenges and testing with Jest.

Congratulations once again on reaching

the end of this course on programming in JavaScript.

You've taken a valuable step on

your journey toward becoming a developer.