COURSE:

INTRODUCTION TO JAVASCRIPT

Index

Module 1 Analyzing the Various Versions of JavaScript

- How to Run JavaScript Programs
- How to Work with Variables in JavaScript
- Understanding the Key Difference Between Var and Let Variables Types
- Guide to JavaScript Hoisting
- Syntax for JavaScript Comments
- Comprehensive List of JavaScript Data Types
- Guide to JavaScript Objects
- How to Check for Data Types in JavaScript
- How to Perform Type Casting in JavaScript
- Working with String Functions - Part 1
- Working with String Functions - Part 2
- Working with String Functions - Part 3
- JavaScript Arithmetic Operators
- Guide to Compound Assignment Operators in JavaScript
- Order of Operations in JavaScript

Module 2 Conditionals

- JavaScript Conditional Section Introduction
- Basic Syntax for Using Conditionals in JavaScript
- Guide to If/Else Conditionals in JavaScript
- Compound Conditionals in JavaScript
- How to Build a Switch Statement in JavaScript to Check for Data Types
- Overview of JavaScript Ternary Operator

Module 3 Functions

- Section Introduction: Introduction to JavaScript Functions
- Basic Syntax for Building Functions in JavaScript
- How Variable Scope Works in JavaScript
- Differences Between Function Expressions and Function Declarations
- How to Work with Function Arguments in JavaScript
- Function Arguments: Reference vs Value
- Introduction to JavaScript Closures
- Introduction to HTML Scripting with Built in JavaScript Functions
- Introduction to the 'this' Keyword in JavaScript
- How to Use the 'this' Keyword in JavaScript Programs

Module 4 Arrays and Data Structures

- Section Introduction: Introduction to JavaScript Arrays
- How to Create Arrays in JavaScript
- Guide to Adding and Removing Array Elements in JavaScript
- How to Use the Splice Function in JavaScript to Remove Specific Array Elements

Module 5 Loops and Iterators

- Section Introduction: Introduction to JavaScript Loops
- Guide to For Loops in JavaScript
- How to Loop Through a JavaScript Object
- Guide to While and Do/While Loops in JavaScript

Module 6 Automating Tasks

- Section Introduction: Introduction to Automating Tasks in the Browser with JavaScript
- Running Google Search Queries in JavaScript
- How to Pull Images from Instagram with JavaScript
- Auto Following Accounts and Hashtags on LinkedIn with JavaScript

MODULE 1.

JAVASCRIPT BASICS

1.1 Analyzing the Various Versions of JavaScript

This guide walks through the various versions of the JavaScript programming language, including topics such as how to lookup browser support.

Before learning javascript, I think it is very helpful to understand the various versions that exist out there. Now we've already talked a little bit about the history of it. So I'm not going to go into that in great detail, but right here if you come to the W3 school's page you can see all of the various versions, all the way starting back in 1997 all the way up through our most recent time.

So we have ECMAScript 1 all the way through 7.

ECMAScript Editions

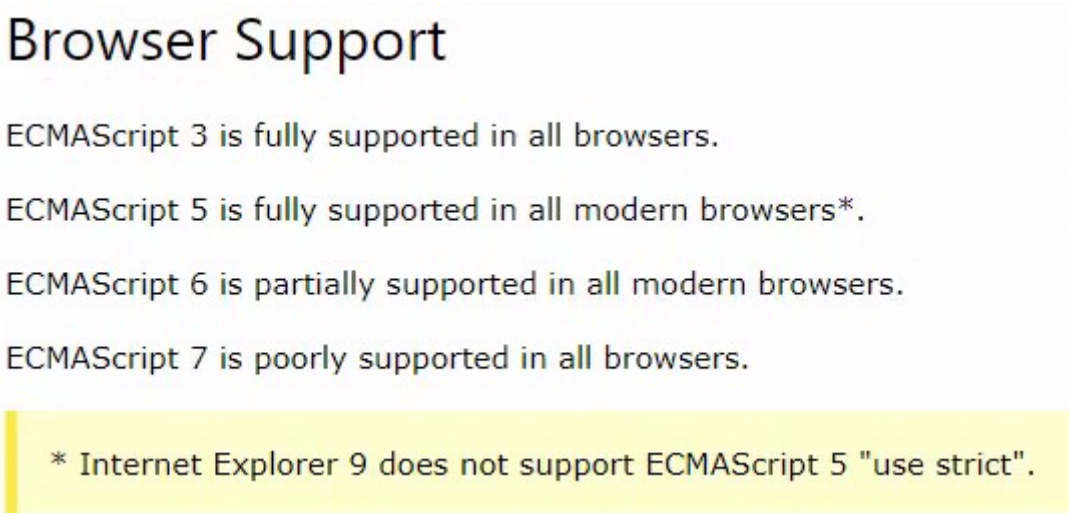
Year	Name	Description
1997	ECMAScript 1	First Edition.
1998	ECMAScript 2	Editorial changes only.
1999	ECMAScript 3	Added Regular Expressions. Added try/catch.
	ECMAScript 4	Was never released.
2009	ECMAScript 5	Added "strict mode". Added JSON support.
2011	ECMAScript 5.1	Editorial changes.
2015	ECMAScript 6	Added classes and modules.
2016	ECMAScript 7	Added exponential operator (**). Added Array.prototype.includes.

ECMAScript 6 is also called ECMAScript 2015.

ECMAScript 7 is also called ECMAScript 2016.

The reason why I wanted to show this page is that it shows some key differences. For example, if you look at the latest version, it has added items such as exponential operators, and one powerful thing was added in 2015. In ECMAScript 6 it added the concept of classes and modules which essentially allowed javascript to function like an object-oriented type language.

Now, this is all very nice and very helpful. It's also important to understand that not every single version is able to be interpreted by every browser.



You have 3, which is supported by all browsers.

5 which is supported in all modern browsers.

That means essentially all the browsers except for some of the older versions of Internet Explorer.

6 is partially supported. I haven't run into too many issues where things such as modules and classes weren't able to be interpreted by the browser.

7 like it said is poorly supported by Chrome, Internet Explorer, Firefox, Opera, all of these have different ways of being able to parse through some of the items available in ECMAScript 7. It's also very important to understand those items. We're going to cover it throughout this course. Different functionalities across all of them and I'll make a note if I'm going to introduce something that may not be able to be interpreted across the board.

Some of these items such as the things available in 6 and 7 are primarily included for the modern javascript framework such as angular or react. In those cases, you don't really have to worry about it because the frameworks themselves will do things, such as preprocess which means that it'll take the code and it will convert it so that it will work in modern browsers. And so that's something that is very nice and helpful to know.

On this page, there are some browser implementations you don't have to worry about. But it's nice to see a history of everything related to javascript. One of the most important things is just this browser support section. (See above Browser Support image)

This is going to be something, that as you get into the more advanced types of builds and projects in javascript. That's where it's going to become more important to understand. But for right now, I essentially wanted to give you more of a practical view on the history that we talked about on a higher level. I wanted to show you the actual version numbers going all the way back to about two decades ago up to modern times.

Version reference guide

https://www.w3schools.com/js/js_versions.asp

1.2 Working with variables

Variables are essential for managing data in JavaScript programs. By understanding how to declare, assign, and access variables, and by grasping the concepts of variable scope, you can write more effective and maintainable JavaScript code.

THEORY:

Variables are fundamental building blocks in programming, serving as named containers for storing data. In JavaScript, variables provide a way to manage and manipulate information within a program.

Declaring Variables

JavaScript offers two primary keywords for declaring variables:

- `var`: This is the traditional keyword for variable declaration. Variables declared with `var` have function scope, meaning they are accessible throughout the entire function in which they 1 are defined.

- `let`: Introduced in modern JavaScript (ES6), `let` provides block scope. Variables declared with `let` are limited to the block of code (defined by curly braces `{}`) in which they are declared. This offers better control and helps prevent unintended variable modifications.

Variable Assignment

After declaring a variable, you can assign a value to it using the assignment operator `=`:

```
var name = 'Kristine';
let age = 12;
```

Multiple Variable Declaration

You can declare multiple variables in a single statement:

```
var name, city, age;
```

This allows for efficient variable definition, especially when you know the names of the variables but don't want to assign values immediately.

Accessing Variables

Once a variable is declared and assigned a value, you can access that value by simply referring to the variable name:

```
console.log(name); // Outputs 'Kristine'
```

Variable Scope

Understanding variable scope is crucial.

- `var`: Variables declared with `var` have function scope. They can be accessed from anywhere within the function they are defined in.
- `let`: Variables declared with `let` have block scope. They can only be accessed within the block of code where they are defined (e.g., within an `if` statement or a loop).

Choosing Between `var` and `let`

In modern JavaScript, using `let` is generally recommended due to its block scoping, which promotes better code organization and reduces potential errors. However, understanding `var` is still important when working with older JavaScript code.

VIDEO:

In this guide, we're going to be introduced to javascript variables. Variables in computer programming are tools that you can use in order to store data. That's one of the easiest ways to think about them. There are a number of analogies that I've heard that are helpful such as thinking of a variable as being like a bucket where you can place contents into that bucket and then from there you can carry it and use it however you need it.

Essentially a variable is like a storage facility for anything that you need to put into it. Now Javascript has all kinds of different reasons to use variables. As you go through this course, we're going to be using them in pretty much every single guide. So it's a good idea to become familiar with them and if you learn any other programming languages, each one of them has some type of a concept of a variable.

In order to be able to access data, to be able to store inside something, and then call it later. That's essentially what variables are. I'm going to show you two different ways that we can use them, usually, I'm only going to use one type of development environment. But just because this is early on in the course I'm going to use both. If I hit (Control + shift + i) this is going to bring up the javascript console right here. If I zoom in and I'm zooming in if you're on a PC by just pressing control and plus or command and plus on a mac. I'm mainly doing this just so that you can see it better.

If I clear out the console, now I can type it. The syntax that I'm going to use is something that I think is intuitive. A variable starts with the letters V-A-R. What we can do in javascript is say var and then the name of the variable. Here I'm just going to say var name and I'm going to set this equal to my daughter's name, I'm going to say var name = Kristine; and then finish off with a semicolon.

If I hit return, this has been stored. The name Kristine has been stored in the variable name. Undefined, what this essentially means is whenever we store a value and we store a piece of data inside of a variable. This just means that nothing else has occurred. In other words, we haven't printed that data out. We haven't sent it in to a function we haven't done anything with it so this return value is just undefined because nothing got returned. If you're new to programming that probably makes no sense whatsoever, and that's perfectly fine. We're going to understand what it means to return something later on.

Don't let that be confusing if you are new to it. Now what we're able to do is, call "name" end with a semi-colon name; hit return and now you can see we have access to this name. That's a very basic way of being able to use this.

Now one other thing just as you're going through the course I'm also going to make all of the code available to you on GitHub. So you're going to be able to reference it here and I'll store it in the guides as well, so you have a reference point.

I'm going to finish off this talk on variables in CodePen. Because I like the way that it renders out, it also makes it easier to see everything. We saw how we could store a name, let's now also talk about how we can store multiple values. I'm going to make one small change here, I'm going to come up to settings and I'm going to come up to behavior.

We talked about javascript and some of the different options. If I click on behavior I don't want this to be auto-saved and I do not want auto-updating preview. It's perfectly fine if you have it. The only thing that I don't like about it, for providing a tutorial and you being able to watch it is on the console. You actually see output maybe sooner than I would want you to be able to see it. It may be kind of confusing. I'm just going to make it so that it doesn't auto-update. You can leave that on your side but I want to be very clear with what every type of code I write does. So I'm going to change that one option, so now that's saved. Now you can see we have this new button here called run.

It's not auto-saving anymore and it's not auto-running. Now we just have to press run and it will run the code for us. Now if I type in the same thing I'm going to say var name = 'Kristine'; I'm doing the same exact code just so you see everything's working. Now if I press run nothing's going to happen because nothing gets returned. This is where code pen and the javascript console are a little bit different. If we want anything to show up down here then we have to do something like this where I say console.log(name); and now if I run this it's going to show Kristine.

Another option just so you know it's available and we'll do it a few times and this course is an alert. What an alert does is for example, if you've ever been on a browser and pressed a button and a little screen came up that said "Are you sure you want to do this" such as if you tried to delete something you may have a little pop up that says "Are you sure you want to do that." Well many times that is a javascript alert.

```
var name = 'Kristine';
```

```
alert(name);
```

you can see that we have a little thing that popped up and it says Kristine because we called alert and name.

Those are two ways of being able to view what we have available.

I'm going to show you a few different ways that we can work with variables. This is a very basic way.

But we also have a different type of variable. We have a type of variable called let. Var is one option let is another and I'm going to change my var name = 'Kristine'; I'm going to say var age = 12; and then let name = 'Kristine'; And now if I say console.log(age); followed by console.log(name); I can hit clear and run this and you can see it prints out 12 and Kristine so everything there is working properly.

Now we're going to discuss in some later episodes what the difference between var and let are. But just to give you a little bit of a heads up those are the two main ways that you will declare variables in javascript, just to give you a little sneak preview. The difference is var is a type of variable that you can change and let is something that you do not want to change and in fact, most versions of javascript won't even let you change it. So in other words, if I say let name = 'Kristine'; I can never change name later on and I'll talk about later on when you want to do something like that.

```
Now there are some other ways that we can work with these. Sometimes
you may not want to actually define the values right away.
city = 'Scottsdale';
age = 12;
```

make sure you end those with semi-colons.

Let me do one more console log, you see that we have age and name. Let's also add city just so you can see that everything's working.

All put together it should look like this

```
name = 'Kristine';
city = 'Scottsdale';
age = 12;
console.log(age);
console.log(name);
console.log(city);
```

Now if I run this, then you can see we have age 12 Kristine and then Scottsdale.

This is another common convention that you're going to see when declaring variables. It's just simply declaring the name. Essentially you're defining and saying OK I have these different data points I want to work with and sometime later I'm going to define them. Now you may wonder why in the world would I ever want to do this.

Well, it's harder to see when you're just doing pure programming like this. But if you imagine a real-world application like say an invoicing application you may want to say that you have things such as a customer name and a total and a list of products or a description or something like that and then you have hundreds of items and you want to render those all on the page.

You may want to actually list each one of those items at the very top and then as your program runs it'll simply fill in each one of the values so it will set each variable equal to that value each time it gets shown in this example for the invoice. That's a time where you'd want to do that just so it

makes it cleaner and you can see in one part of your code all of your variables, where they're listed out and then you can go and you can access them whenever you need to.

That is how you can work with variables in javascript.

```
var name, city, age;
name = 'Kristine';
city = 'Scottsdale';
age = 12;

console.log(age);
console.log(name);
console.log(city);
```

Coding Exercise

Create 2 variables, called cat, and dog respectively, and set their values to something.

1.3 *var* vs. *let*: Key Differences in Variable Declaration

The introduction of `let` in ES6 addressed the scoping limitations of `var`. By using `let`, you can write cleaner, more maintainable, and less error-prone JavaScript code. Focusing on the unique feature of 'let' that doesn't allow it to be re-defined.

THEORY:

In JavaScript, `var` and `let` are both used to declare variables, but they differ significantly in their scoping rules, which determine where a variable is accessible within your code. Understanding these differences is crucial for writing maintainable and error-free JavaScript.

`var`: Function Scope

- Variables declared with `var` have **function scope**. This means they are accessible throughout the entire function in which they are declared, regardless of block boundaries (code enclosed in curly braces `{}`).

- `var` allows **re-declaration** of the same variable within the same scope. This can lead to unexpected behavior and make code harder to debug.

`let`: Block Scope

- Variables declared with `let` have **block scope**. They are only accessible within the block of code where they are defined. This provides better control over variable visibility and reduces the risk of accidental modifications.

- `let` prevents **re-declaration** of the same variable within the same block. This helps avoid errors and makes code more predictable.

Why `let` is Preferred

In modern JavaScript, using `let` is generally preferred over `var` due to its block scoping. This offers several advantages:

- **Reduced Errors:** Block scoping helps prevent accidental variable overwrites, making code more robust.
- **Improved Code Organization:** Block scoping allows for more organized and modular code, as variables are confined to their respective blocks.
- **Alignment with Modern Standards:** `let` is part of the ES6 (ECMAScript 2015) standard, which introduced many improvements to JavaScript.

When to Use `var`

While `let` is generally preferred, there might be situations where you encounter `var` in older codebases. Understanding how `var` works is essential for maintaining and working with such code.

Example

```
function example() {
  if (true) {
    var x = 10; // Function scope, accessible throughout the function
    let y = 20; // Block scope, accessible only within the if block
  }
  console.log(x); // Outputs 10
  console.log(y); // Error: y is not defined
}
```

VIDEO:

Now that you have a decent idea of how to create variables, both how to define them and also how to set their values. Let's talk about how we can actually change a variable's value. This is going to lead us directly into the conversation for understanding the main difference between the var type of variables and the let type of variables that we just walked through.

I'm going to create a new variable here and this one I'm going to call age and we'll set it equal to 12 var age = 12; and then just to make sure everything's working. You say console.log(age); If I run this it runs it and it's 12.

Everything is working just like how you'd expect. Now I can redefine this so if I come up here, copy it and actually I'll copy both lines. Now if I want to redefine this and say change it to 15 and run it again you can see that we have 12 and 15 both printed out here perfectly.

So everything there is working the way that you may have expected. You hit clear and also save it. And now let's talk about how LET works differently. I'm going to change this to let instead of var.

Now let's try running it exactly the same way we did. If I hit run you may notice nothing is happening. Hit it again. Still nothing happening. That's not a problem with your browser or with code pen this is how the let variable is supposed to work. You are not supposed to be able to redefine the entire variable.

And if you want a little bit of insight on that if you come up and click on this little arrow here

this is where we talked about analyzing JS. If you click on that you can see that it actually shows you the problem and it says that age has already been declared.

It also says to google it which would give you a little bit of an idea on exactly what is happening. You can do that if you feel like it. I'll simply explain the way that I personally like to think about it. And that is that a var gives you a very flexible type of variable you can think of it almost like a temporary type of a container. You can use it you can get rid of it. You can replace it, override, everything works exactly the way that you'd expect that to work.

let on the other hand gives a little bit more of a strict framework for how you can redefine variables and you may wonder when you would ever want to use that. I can tell you there are a number of times where I've accidentally overridden variables. So I define something like say that I had a blog and I had a variable called blog posts and that variable was supposed to contain the blog post. But at some other part of the program I wanted to call something else blog post I redefined it and accidentally overrode that value. If I would have used a let then that wouldn't have happened.

The program would have thrown an error and I would have seen where the problem was so whenever you want to have some type of a variable or a data point that you don't want to have it accidentally overridden, that's where let comes in very handy and you see quite a bit of modern javascript is going to use let throughout the programs and that also extends into the frameworks like angular JS.

You see a lot of let base variables because it's a little bit easier to control and you don't have as much of that fear of having one of your values accidentally overridden later on in the program. So that is an introduction to the main differences between var and let

And also it also demonstrates how you can change variable values.

Var variables can be re-defined

```
var age = 12;
console.log(age);

var age = 15; // yep!
console.log(age);

Let variables cannot be re-defined

let age = 12;
console.log(age);

let age = 15; // nope!
console.log(age);
```

Coding Exercise

Reassign the let variable below so it equals 10

```
let variable = 10;
```

1.4 Guide to JavaScript Hoisting

Hoisting is a fundamental concept in JavaScript that affects how variables and functions are scoped. Understanding hoisting can help you write more efficient and error-free code. However, adhering to best practices and declaring variables at the beginning of their scope can improve code clarity and maintainability.

THEORY:

In JavaScript, hoisting is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation 1 phase. This means that you can use a variable 2 before it has been declared in your code

How Hoisting Works

1. **Variable Declarations:** When the JavaScript engine encounters a `var` declaration, it hoists the declaration to the top of the current scope. However, only the declaration is hoisted, not the initialization. This means that the variable exists but has an initial value of `undefined` until the assignment is executed.
2. **Function Declarations:** Function declarations are also hoisted. The entire function is moved to the top of the scope, allowing you to call the function before its declaration in the code.

Example:

```
console.log(myVar); // Outputs: undefined
var myVar = 5;
```

In this example, the `var myVar` declaration is hoisted to the top, but the assignment `myVar = 5` remains in its original location. Therefore, when `console.log(myVar)` is executed, `myVar` exists but holds the value `undefined`.

Caveats of Hoisting

- **Only declarations are hoisted:** Initializations are not hoisted. If you try to use a variable before both declaration and assignment, it will result in an error.
- **Hoisting can lead to confusion:** While hoisting can be helpful, it can also make code harder to read and debug, especially for developers unfamiliar with this JavaScript quirk.

Best Practices

To avoid potential issues and write cleaner code, it is recommended to:

- **Declare variables at the top of their scope:** This makes the code more readable and predictable.

- **Use `let` and `const`:** These newer keywords have different scoping rules and are less prone to hoisting-related issues.

VIDEO:

We've talked about variables. How to define them, how into reassign values, we've talked about the differences between the let and var, and now we're going to talk about something that is pretty much specific to javascript.

Very few programming languages work with this topic the way that javascript does. The topic is called hoisting. Hoisting is essentially a very specific way that the javascript interpreter actually reads variables. So we're going to go through some examples to see how this works. What this is going to essentially be related to, is where you should define your variables in your program. If you define them in the wrong order then you might run into some odd types of bugs. The way that hoisting works is, I'm not going to declare a variable I'm just going to assign a variable. If I have a variable like this name = 'Kristine'; and type out console.log(name);

Notice I've not used var or let but with Hoisting what I can actually do is say var name; at the bottom. If I run this code it prints out Kristine.

There is no issue with that whatsoever, the reason for this is something that may seem a little bit confusing if you're brand new to programming, or if you're coming from a different programming language and you've never used javascript before. Essentially what the interpreter does with hoisting is it goes and it tries to find every spot where a variable has been declared.

It looks for all of these kinds of things like var name; and it actually preloads this at the very top. Writing this code is really viewed more like this :

```
var name;
name = 'Kristine';
console.log(name);
```

Because of hoisting and that's the reason why it's called hoisting because these declarations get hoisted up to the top. That is the way it works.

There is a little caveat to hoisting and that is it only works with declarations. It does not work with assignment. I'm going to try to do something kind of similar. I'm going to say console.log(age); and then below. I'm going to say var age = 15;

```
console.log(age);
var age = 15
```

Now if I come here and run this. You can see it doesn't print out 15. It prints out undefined. If you've never worked with hoisting before, javascript, that may seem a little bit weird because it kind of looks like we declared the age variable but actually we declared it and assigned it to a value at the same time.

Whenever you do that is viewed differently by the javascript interpreter compared with if you just defined it. Or if you just declared it these do not get hoisted up to the top. That is a very key difference to keep in mind. And it leads to one of the first best practices that we're going to talk about and that is to avoid hoisting issues. It's really considered the best practice to assign your variables and to declare them at the very top.

Doing something like that could lead to some weird bugs. It's always better to write your code more like this:

```
var age = 15;
console.log(age);
```

Because if you do this and run it now you can see it prints out 15 exactly the way it should.

Hoisting is something that is very important to know and it's important to know that that is the way that javascript views declarations but to make things easier to make your code easier to read and also easier to debug the best practice is to always list your variables right at the top of your programs.

```
// Hoisted
name = 'Kristine';
console.log(name); // 'Kristine'
var name;

// Initializers not hoisted
console.log(age); // undefined
var age = 33;
```

Coding Exercise

Create a variable called name and its value must be set to "Jordan".

1.5 Syntax for JavaScript Comments

This guide explains both syntax options for adding comments to JavaScript files.

THEORY:

Comments in JavaScript are lines of code that are ignored by the interpreter. They serve as annotations and explanations for human readers, improving code readability and maintainability.

Types of JavaScript Comments

JavaScript supports two types of comments:

1. **Single-line Comments:** These comments start with `//` and continue until the end of the line. They are used for brief explanations or annotations.

```
// This is a single-line comment
console.log("Hello, world!"); // This comment explains the
following code
```

2. **Multi-line Comments:** These comments start with `/*` and end with `*/`. They can span multiple lines and are used for longer explanations or to comment out blocks of code.

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

Best Practices for Using Comments

- **Clarity and Conciseness:** Comments should be clear, concise, and easy to understand. Avoid unnecessary or redundant information.
- **Focus on "Why", not "What":** Comments should explain the purpose and reasoning behind the code, not just what the code does. The code itself should be self-explanatory for the "what".
- **Keep Comments Updated:** Ensure comments are updated when the code changes to avoid inconsistencies and confusion.

- **Avoid Over-Commenting:** Comment only when necessary. Well-written code should be self-documenting, minimizing the need for excessive comments.
- **Use Comments for Documentation:** Comments can be used to generate documentation using tools like JSDoc.

When to Use Comments

- **Explain complex logic:** Clarify intricate algorithms or non-obvious code segments.
- **Provide context:** Explain the purpose of a function, class, or module.
- **Document APIs:** Describe the parameters, return values, and usage of functions and methods.
- **Comment out code:** Temporarily disable code blocks for testing or debugging.

Self-Documenting Code

While comments are valuable, strive for self-documenting code by:

- **Using descriptive variable and function names:** Choose names that clearly indicate their purpose and functionality.

- **Following consistent coding conventions:** Adhere to established style guides for formatting and structure.

- **Breaking down complex code:** Divide lengthy code into smaller, more manageable functions or modules.

By combining clear comments with well-written code, you can create software that is easier to understand, maintain, and collaborate on.

VIDEO:

In this guide, we're going to talk about comments in javascript. Every programming language I've ever worked with has a concept of comments and what comments are is they allow you to put text in your code without actually having it run. And so what will happen is you can type anything you want in these comments and then when the javascript interpreter scans through the code it recognizes what a comment is and then it completely ignores it

Anything you put inside of it is not going to run. Before I show you the syntax for this I want to make one statement about code comments and this is something that I've seen through the years. And so I want to introduce it now. And that is that you need to be very intentional about your code comments. Because I have seen many developers especially new developers that load up their programs with all kinds of comments and what I've seen also happen is as that program develops

and as they change functionality in their programs they don't always go back and update the comment.

In other words, if they say that the comment is, that a function is going to do x y z type functionality and it requires these arguments and they put that in the comment, then if they make a change and the behavior of the function changes but they don't update that then when someone else goes or even when they go back and they look at the comment later on it's going to be confusing because they're going to try to follow those instructions and they're not going to fit the updated version of the code.

So be very careful when you're typing them in. I actually rarely put comments into any code that I write and I follow the philosophy that my classes my modules my functions should be self-describing and you'll see that as we start to get into more advanced topics. I like to use very descriptive words for variables and for functions so that I don't need comments. I can actually read what I've named something and then that describes what it does.

That's a little kind of a best practice when we're talking about how we organize code and how we name things that may not seem like a big deal. But the more advanced a project gets the more important naming gets. And you don't want to rely on comments because those could say anything and they can be very confusing and lead to some bugs if they're not updated.

With all that being said now let me show you actually how to use comments. There are two different ways you can do it, the first way is a single line comment. With that you just do two slashes // and then you can put anything you want after that for example.

```
//this line of code is commented out
```

As you can see the syntax highlighter shows that this is grayed out which represents that nothing is going to happen if I run this. Nothing comes out in the console literally no code gets changed whatsoever. Now that is when you have just one line of comments. Now the cool thing about this as well is you can actually put them on the same line as code as long as you put it after the code.

```
If I do console.log('Hi there'); // Here is another comment
```

You'll see this as you go through documentation and as you get further along on your javascript journey you'll see this happen quite a bit where a developer will have a single line of code and then they want to describe something that is just on that line. And so this is a good way of doing it. Now another way if you want to have multi-line comments is to use this syntax, a slash followed by an Asterix

```
/* and then you can
put anything you
want here on
multiple lines
*/
```

and then you close it off with an Asterix and other slash and then everything inside of there is going to be contained in that comment and nothing is going to happen. If you run this entire program right here you can see the only thing that gets run is the one console.log line of code. Everything else is considered a comment. That is the syntax for both types of comments that you can use in javascript.

```
// You can put anything you want here

console.log('Hi there'); // Here is another comment

/*
Anything
you want
here
*/
```

1.6 Comprehensive List of JavaScript Data Types

This lesson walks through each of the JavaScript data types. Additionally, we'll discuss how JavaScript utilizes dynamic typing.

THEORY:

In JavaScript, data types classify the different kinds of values that can be used and manipulated in your code. Understanding these data types is crucial for writing effective and error-free JavaScript programs.

JavaScript is a dynamically typed language, meaning you don't need to explicitly declare the data type of a variable. The JavaScript engine determines the type automatically based on the value assigned to it.

Here's a breakdown of the primitive data types in JavaScript:

1. **Boolean**
2. **Null**
3. **Undefined**
4. **Number**
5. **String**
6. **Symbol (Since ECMA 6)**

Dynamic Typing in JavaScript

Unlike statically-typed languages (like Java or C++), JavaScript doesn't require you to specify the data type when declaring a variable. This offers flexibility but also requires careful attention to avoid unexpected type-related errors.

Example:

```
let age = 25;      // age is a number
age = "twenty-five"; // age is now a string
```

In this example, the variable `age` initially holds a number. Later, it's assigned a string value. JavaScript allows this dynamic reassignment of types.

Benefits and Drawbacks of Dynamic Typing

- **Benefits:**
    - Increased flexibility and faster development.
    - Less code required due to no explicit type declarations.
- **Drawbacks:**
    - Potential for runtime errors due to unexpected type changes.
    - Requires careful testing and debugging to catch type-related issues.

This comprehensive overview provides a solid foundation for understanding JavaScript data types. As you progress in your JavaScript journey, you'll learn how to leverage these data types effectively to build complex and dynamic applications.

VIDEO:

In this guide, we're going to talk about javascript data types. If you are new to javascript development or new to programming, then let's talk about what a data type is. Essentially it is how javascript categorizes all of our data points. If we have variables a data type is a way that javascript sees that type of variable so does it see a sentence or does it see a number. The reason why that's important (there are more categories than that). Those are just a few examples.

Imagine that you have a sentence of words that sentence can have certain types of functions called on them so you could capitalize on all the words if you tried to capitalize a number. Then an error is going to happen and it should because we need to know and we need to be able to use the right functions on the right type of data and that is what the data types allow us to do.

We are going to go through each one of the data types provided by javascript. I'm going to put a comment at the top for each one.

The first one, we're going to start alphabetically, is the Boolean data type.

Now Boolean can have two potential values. True and false.

I can declare some variables.

```
var truthy = true;
var notTruthy = false;
console.log(truthy);
```

If I run this you can see it returns true.

Now, this is not a name it's not a letter it's not a series of letters like our name variables are. These are just two values. True and false. And so those are the only two things that a boolean can be. So this would be something like saying is a user a paid user if you're building an application and it's a SASS product and you want to be able to see which of these users is paid and which ones are a free member.

Well, you can in your data say this is a paid user it's true that they are paid and then when we get into conditionals we're going to talk about how we can leverage that to give our programs dynamic

behavior. Right now just know that we have a boolean data type and it has two potential values true and false.

The next one we're going to talk about is the ***Null*** **data type.**

Now null can only have one value and that is null.

If you say

```
var nully = null;
console.log(nully);
```

If I run this then it just prints out null.

Essentially what null is, is it's the absence of any kind of value.

The next one we're going to do you've kind of already seen but it's important to understand that it's different than null. So null means empty. So this would be something like say, that you had a registration page and you had some optional fields such as a Twitter user name or something like that. If a user decided not to fill that in then that value for that Twitter user name would just be null. It's our way of understanding that that variable or that value could be there. But in this case, it's just empty.

Moving down we're going to go to the next one which is undefined. We've seen that a few times and so undefined allows us to do is something that usually you're going to be using more with debugging because it's exactly what you get when something is simply declared. And it's not given a value.

So, here:

```
var notDefined;
console.log(notDefined);
```

Just saying, var not defined set it equal to, actually in this case I'm just we're not going to set it equal to anything because this is usually the way you're going to see this.

If I hit run you can see it's undefined. I didn't set it equal to undefined because by default when you simply declare a variable javascript is going to set it equal to undefined

That's kind of an important thing to know because there are going to be many times when you're debugging your program and you'll think that you have access to some value or you think that you set the value only later on to find out. It comes back as undefined which means at some point the spot where the value got assigned got skipped or it never happened. Something like that so that's why that's important.

And it's also nice because you can do things like check to see is this value defined yet or not. And there may seem to be some similarities between null and undefined but hopefully you can kind of see that subtle difference where null means that it is defined. It does have access to that variable but it just is empty. There's nothing there whereas with undefined what it means is it has not been assigned yet so it's not empty. There literally is no value that is there yet it needs to be assigned later.

Now the next one we're going to do may seem a little bit more practical and that is number.

We've seen this before this is where we can do something like

```
var age = 12;
console.log(age);
```

This is just going to print out that value. 12 everything there works.

Now, this is where I want to take a little bit of a pause and talk about how javascript works with variables and data types much differently than many other programming languages. I

f you're coming to javascript from say, Java or C or C++ those are called statically typed languages. In those languages, what we just did here would not work.

The reason it would not work is that those languages typically require you to define what the data type is going to be. So the syntax would look something like this

```
var age : Number = 12;
```

And then in those cases those compilers when it's scanning the code and it comes to this age variable it says OK we can expect that age is going to be of the number data type. And here is the value. Well, javascript is similar to languages such as Ruby and Python where it actually skips that step. So you don't have to type that in the parser actually does that work for you. So in this case the interpreter is going to hit line 12.

It's going to see var it's going to see age and then it's going to see that we assigned it. Now it does some checking and it checks to see what data type is 12 and when it sees that data type the data type for 12 is number it does the assignment and it forces that.

Javascript has some kind of side languages or some precompiler such as typescript that are very similar to javascript but they force you to put in the data types that are actually the reason why typescript is called typescript. A lot of it is a personal opinion.

There is a pretty big debate in the developer community on if statically or dynamically type languages are better people who believe in statically type languages, think that it's much better because it enforces this. Imagine a scenario with age where we think it's going to be a number but there are times where somebody types it in and it's a string.

Then if we ever tried to do any type of computation on that. Say we try to add age plus another number it's going to throw an error and the program is going to crash whereas if we had to enforce that.

Then the program wouldn't have even compiled in the first place. Now some of that is kind of high level and starting to get a little bit more advanced so don't let that scare you off. I just want to give you a little bit of an idea of data types in javascript and how they work differently than in many other languages.

Continuing down the line. We have the

String data type now strings are what you're usually going to see used for words and sentences.

```
var name = “Kristine";
```

And this is using quotes. And then if I come here just like we've done before console.log this it's going to print out Kristine.

With the string data type we have a couple of different options we can do this.

But we can also do something like this as well.

```
var name = "Kristine";
var nameTwo = 'Jordan';
console.log(name);
```

If I do name two. Notice that this is going to work exactly the same way. So we have name. And if I get a name two and run it it's going to print out both of these properly.

```
var name = "Kristine";
var nameTwo = 'Jordan';
    console.log(name);
    console.log(nameTwo);
```

One of these had double quotation marks and one of them had single. For the most part in how you're going to be using javascript, it is not going to matter one way or the other. It starts to become more important when we talk about formatting output but we'll talk about that later on when we get into an entire module just dedicated to the string data type.

For right now just know that both of these when the javascript interpreter sees that it's going to recognize quotation marks and it's going to set that equal to the string data type.

The last one I'm going to talk about is a little bit more advanced and we're not going to go into a lot of detail on it mainly because you in building introductory type programs will never even touch the symbol data type. This is something that is brand new in ESX. So anything prior to that it will not know what a symbol is.

Essentially what symbols do is they're kind of similar to strings except they have some very specific rules about them. They can not be changed. There can only be one of them. And so they're the closest thing that Javascript has to what's called an immutable type object. So when you create it then it is what it is you cannot change that value.

It's used primarily for working with objects which is what we're going to be talking about next. But they're more for advanced type features but I'm still going to show it to you just so you can see what all of the types of variables look like.

```
var mySym = Symbol();
console.log(mySym);
```

It's just going to print out an object symbol. Now you can pass a string in here so I could just pass foo

```
var mySym = Symbol('foo');
console.log(mySym);
```

And one thing for new developers as you're reading documentation. You're going to see foo and bar used a lot. And I've actually had developers come up and ask what is special about foo and bar. As a side note, there is nothing special about them.

They for some reason have just been used for decades in programming documentation and in explanations because it's a very quick way to just know a simple word to use it is literally the exact same.

I could type *asdf* or something like that it doesn't represent anything special. Whenever you see that on a stack overflow or some piece of documentation know that all they're doing is they just wanted a simple word to use. The only reason I wanted to bring it up is that I have been asked that by students multiple times. There's nothing special about it. It's just a piece of kind of programming lore in that sense.

If I hit run now. Now we have an object symbol, but inside of it is the actual foo. If we're to use this in an object which we're going to talk about next then it's going to have unique representation which is going to be the symbol of foo.

That is the full set of data types in javascript. Once again that's boolean null undefined number string and symbol in the next guide we're going to talk about the very important type of component in javascript development that we're going to be using out this entire course so I wanted to dedicate an entire lesson to it. And that is the object.

```
// Boolean
var truthy = true;
var notTruthy = false;

// Null
var nully = null;

// Undefined
var notDefined;

// Number
var age = 12;

// String
var name = "Kristine";
var nameTwo = 'Jordan';

// Symbol
var mySym = Symbol('foo');

console.log(mySym);
```

Coding Exercise

Assign the variable a value of data type string.

```
let array = [];
```

1.7 Guide to JavaScript Objects

This guide explains how to work with JavaScript objects. This will include: how to define an object, how to parse values from an object, how to work with nested objects, and how to add new key/value pairs on the fly.

Theory:

Objects are essential building blocks in JavaScript, providing a structured and flexible way to represent and manipulate data.

Their versatility and dynamic nature make them indispensable in a wide range of JavaScript development scenarios. As you progress in your JavaScript journey, understanding and mastering the use of objects will significantly enhance your ability to build robust and efficient applications.

In JavaScript, objects are fundamental data structures used to store collections of key-value pairs. Each key, a string (or symbol), uniquely identifies a specific value within the object.

These values can be of any JavaScript data type, including numbers, strings, booleans, functions, and even other objects. This flexibility makes objects incredibly versatile and widely used in various JavaScript applications, from simple scripts to complex frameworks.

Object Syntax

Objects are defined using curly braces `{}`. Within these braces, key-value pairs are specified, separated by colons `:` and commas `,`.

```
var user = { name: 'Kristine', age: 12, city: 'Scottsdale' };
```

Accessing and Modifying Properties

Object properties can be accessed using dot notation `.`:

```
console.log(user.name); // Outputs 'Kristine'
```

Modifying properties follows the same syntax:

```
user.age = 13;
```

Nested Objects

Objects can be nested, meaning an object can hold another object as a value:

```
var user = {
  // ... other properties
  grades: {
    math: 90,
    science: 80,
    languageArts: 100
  }
};
```

Accessing properties within nested objects involves chaining dot notation:

```
console.log(user.grades.math); // Outputs 90
```

Dynamic Object Manipulation

A key strength of JavaScript objects is the ability to dynamically add or modify properties during program execution:

```
user.grades.coding = 99;
```

This dynamic nature makes objects adaptable to changing data and program requirements.

Video Dialogues:

This is going to be a very exciting lesson. The reason for it is because we are going and talk about javascript objects now. As you go along your javascript journey you are going to discover you're going to be using objects all day long.

Objects are used for all kinds of different programs for all kinds of different frameworks for working with functions we're working with each one of the variables data structures everything like that.

You're going to use objects for and the reason for it is because it has one a very nice syntax but it also gives you an ability to query data and to set data and also to create a blueprint for your object. And that may sound a little bit abstract and so we're going to go through a basic example.

I'm going to create a variable here called user and the syntax for this is to use curly braces. And so I'm going to say var user and inside of it I'll just go name and then it'll go Kristine. And that is all I have to do.

```
var user = { name: 'Kristine' }
console.log(user);
```

So if I console log this. Notice also I didn't put a semicolon after `var user = { name: 'Kristine' }`. Now if I run this it's going to print out the object. Now that doesn't seem like a big deal but that means that our syntax is right. But now I can do object or user name so I can actually call this key inside of it. It will look like this

```
var user = { name: 'Kristine' }
```

```
console.log(user.name);
```

And now you can see it prints out Kristine.

This gives us the ability to call with this decimal type notation and actually call and reference points inside of the object. This is something that is very powerful and let's say that we want to rename something.

```
var user = { name: 'Kristine' }
user.name = 'Jordan';
console.log(user.name);
```

Now if I run this, you can see that that object has been changed.

What I did was I reset the name and got rid of Kristine put Jordan instead called it.

Now we're referencing that new name. That's a very basic way of working with objects. Now let's get into a little bit more of an advanced example. The syntax whenever you're defining just a single key-value pair and that's what you have here. You have a key on the left-hand side followed by a colon followed by whatever the value is.

Whenever you have just one it's fine for you to put it on one line. But when you have multiple then usually you're going to want to put this on multiple lines just like we're doing in the example below. Now, as you add more items or as you add more key-value pairs put a comma at the end and then add your next one. Here I'm going to say age is equal to 12 and city is equal to Scottsdale.

Now I can reference age. Run this again and it returns 12. We could also set city this same exact way. I won't go through that because that's what we just did. But you can play around with these so you can get familiar with the syntax.

Essentially what this is doing is it's giving us the ability to create an object with multiple values. Each one of those values has something inside of it that we can reference whatever name that we choose to give it. This makes for a really nice interface and like I mentioned at the beginning you're going to be using this syntax all day long as you build javascript programs.

Now in addition to this, this is a very basic type of object. Now let's get into a little bit more advanced type work. If I do a comma after Scottsdale I'm going to show how we can actually have nested objects so you can put an object inside of another object.

This is something that is very commonplace for things like API development where you need to be able to group items inside of other items. Here I'm going to say grades and from there we're going to use the same exact syntax so I'm going to use curly braces again. We started with curly braces when we assign the variable and then we put our values grades is still just a key for our user object. But now grades has semicolon but instead of something like a string or a number we're putting another object right there. Here I can put more keys and more key-value pairs so I can say math and we'll say 90 and then science and we'll call 80 and then Language Arts we'll put it at a hundred.

The way that you'd reference this may seem intuitive but maybe not depending on your experience with it. But just like we able to call age if we want grades all we have to do, is traverse the object so we're calling grades.

If I were to console.log that it's going to print out each one of those grades. Now if I want a specific grade it's the same exact way we picked up age. So if I come here and now I want to grab math I just type drop math hit run and it prints out 90 which is the value for math.

Notice what we did there, we started at the user object then we looked for grades which also contains an object inside of it as its value. And then we grabbed math and we finally got the value we wanted. Now, this is cool but it gets even better. One thing that happens a lot is as you're building objects you need to dynamically build them on the fly. Let's say that Kristine got a new class and so we need to add a grade dynamically. We can't come back in and change what's already there. We need to actually update it which is something that could happen in a program.

What I can do is say `user.grades.coding = 99;` What this is going to do is it's going to add a new key-value pair inside of grades. In order to add something we don't have to come back into the user variable and add it, we can actually do it at runtime.

I think that's something that is pretty neat. Now, if I say user grades and if I run this you can see right here, that now our object of grades has coding language, arts, math, and science even though it didn't start out with it. We're able to define that and add it as our program was run. That is something that one it's pretty cool but also the further you go along in your javascript development journey you're going to see objects used all over the place so this is something I definitely recommend for you to experiment with.

Build your objects. Try to build all kinds of different combinations of putting objects inside of other objects finding out how to parse them. And that is going to help you out quite a bit as you go through this course as well as when you start to build out real-world javascript programs.

```
var user = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale',
  grades: {
    math: 90,
    science: 80,
    languageArts: 100
  }
}
user.age = 13;
user.grades.coding = 95;
```

Coding Exercise

update the user grades to add an english grade that is 87

```
var user = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale',
  grades: {
    math: 90,
    science: 80,
    languageArts: 100
  }
};
```

1.8 Determining Data Types in JavaScript

Checking for a component's data type in JavaScript may be very important since the data type determines the types of functions that can be passed.

The `typeof` operator is a simple yet powerful tool for determining the data type of a variable in JavaScript. By understanding and utilizing `typeof`, you can write more reliable, predictable, and maintainable code.

THEORY:

In JavaScript, understanding the data type of a variable is crucial because it dictates the operations that can be performed on that variable. This is essential for writing robust and predictable code. JavaScript offers a simple mechanism to check data types: the `typeof` operator.

The `typeof` Operator

The `typeof` operator returns a string indicating the type of the operand. Here's how it works with different data types:

- **Numbers:** `typeof 12;` returns `"number"`.
- **Strings:** `typeof 'Astros';` returns `"string"`.
- **Booleans:** `typeof true;` returns `"boolean"`.
- **Objects:** `typeof { name: "Kristine" };` returns `"object"`.
- **Undefined:** `typeof age;` (where `age` is an undeclared variable) returns `"undefined"`.
- **Null:** `typeof null;` returns `"object"` (this is a known quirk in JavaScript).

Practical Applications

Knowing the data type of a variable is essential for several reasons:

- **Function Calls:** Different functions expect arguments of specific data types. `typeof` helps ensure you're passing the correct types.
- **Conditional Logic:** You can use `typeof` in conditional statements to execute different code blocks based on a variable's data type.

- **Data Validation:** When receiving data from external sources (like user input or APIs), `typeof` helps validate the data before processing it.
- **Debugging:** `typeof` is a valuable debugging tool for identifying unexpected data types that might be causing errors in your code.

Example Scenarios

- **Handling Optional Values:** In a web application, a user's Twitter username might be optional. You can use `typeof` (or checks for `null`) to determine whether to display a link to their Twitter profile.

- **Data Type Conversion:** If a variable is expected to be a number but is received as a string, you can use `typeof` to detect this and convert the string to a number before performing calculations.

VIDEO:

Now that we've discussed variables and the various data types related to javascript. Let's see how we can find out what kind of data type a specific element is. This may not seem like the most important thing but you will actually see as you get further down in the course and also as you build out projects that this is incredibly important because you need to be able to know what type of object you're working with, what type of data type you're working with.

Because if you don't know that then you're not going to know what types of functions you can call on them which is something you're going to be doing pretty much all day long. So let's talk about how we can do that. I am not using CodePen for this one. I'm using the regular javascript console in the browser's developer tools. The reason for that is because I like how it shows returned elements not just the items that get printed out. I don't want to console log each one of these. In order to find this out see what type of data type something fits into, there is the type of function, you can pass something to it. Right here I can pass 12, run it and you can see that it tells us that 12 is a type of number.

```
typeof 12;
"number"
```

Now if I pass in a string

```
typeof 'Astros';
"string"
it will tell me that this is a string so this is a way of being able
to quickly check to see what type of data element something is.
If we want to keep on going down the line of some of the other ones if
you want to check to see if something is a Boolean, you do type of
true and it'll tell you Boolean.
typeof true;
"boolean"
```

We could go down the list on each one of these. I'm only going to do one more which is the object. If I do

```
typeof { name: "Kristine" };
"object"
```

this is going to tell me this is an object.

```
You can see that we have a number, a string, a boolean, and an object.
This is something that's pretty helpful but it probably doesn't make a
ton of sense on why you'd want to use that. So let's go through a
little bit of a more practical example. Right here, I can say
javascript
var age;
undefined

you can see that this is undefined.
```

Now imagine a program where all the code isn't right in front of you because many times you're going to be using code libraries and you're not going to have the nice list of every one of the variables or functions you have access to and you may also not know what type of data they're returning. Because imagine one scenario where age gets sent to you as a string. So in other words it might be 12, but it might be 12 represented as a string, or it could be an integer like our example or the number data type.

That is something that is very important to know. Now that we have our variable, pretend it's way up in some other file we can't even see it. We can run a program and say

```
typeof age;
"undefined"
```

Right now, it's undefined. Undefined is a type we can check to see has something been set. If not it's going to be equal to undefined and then we can go and see what is there. Another thing you could also do is say that you wanted to see if there were any values inside of it you could do.

```
Is set age = null
Now if I say
typeof age;
"object"
```

you can see now it's an object.

That is something to keep in mind that null is going to return object, in this case, it was stored in the age variable a good example of that would be, imagine that you know a web application and you weren't sure if all of the values that were coming back are going to be full.

You knew what you had access to, but you didn't know which elements actually had something. For example, I talked about if you had a web application and one of the attributes was a Twitter username but it's optional, it may or may not have information. If it does have it you may want to do something like make it clickable and go to the Twitter profile page. If it doesn't exist you don't want the link to show up at all.

In that case, you can check to see if it's this value is undefined, or in this case, is that value null? If it is then I don't even want you to show that element whatsoever. That's just kind of one very basic example of why this high level of checking can be important, and when it can be very helpful. Another would be the age example that I gave earlier where age might sometimes be a string, or sometimes might be a number depending on what its data type is. That's going to determine what types of functions you can pass to it.

That is something that is very important to know and good to keep in mind it's starting to get into things that are slightly more advanced but I don't want to go through this without giving you an idea on how you can check for data types because later on in the course when we do, do that kind of thing I want you to already have an introduction to it in your mind. So that is how you can check for the data type of any element inside a javascript.

```
typeof 12;
// "number"

typeof 'Astros';
```

```
// "string"

typeof true;
// "boolean"

typeof { name: "Kristine" };
// "object"

var age;
// undefined

typeof age;
// "undefined"

age = null;
// null

typeof age;
// "object"
```

Coding Exercise

Get the type of the variable

```
var variable = 20;
```

1.9 Type Casting

This lesson examines how to convert data from various data types into different data types in JavaScript. Additionally, we'll discuss the issues related to automatic type casting and when this type of feature is required in real world development.

THEORY:

Type casting, also known as type conversion, is the process of converting data from one data type to another. In JavaScript, this is a common practice due to its dynamic typing system, where variables can hold values of different types.

Automatic Type Casting (Implicit Conversion)

JavaScript often performs automatic type casting, attempting to make sense of operations involving different data types. While convenient, this can sometimes lead to unexpected results.

Examples:

- **Successful Conversion:** `'100' - 42;` // Returns 58 (string converted to number)
- **Unexpected Behavior:** `100 + '42';` // Returns "10042" (number converted to string for concatenation)

Manual Type Casting (Explicit Conversion)

To avoid ambiguity and ensure predictable outcomes, you can perform manual type casting using built-in JavaScript methods and operators.

Methods for Type Casting

1. **Converting to String:**
   - `String(value)`: Converts any value to its string representation.
   - `value.toString()`: Converts a number to its string representation.

```
let age = 12;
String(age); // '12'
age.toString(); // '12'
```

2. ## Converting to Number:

   - `Number(value)`**: Converts a value to a number.**
   - **`parseInt(value)`:** Parses a string and returns an integer.
   - **`parseFloat(value)`:** Parses a string and returns a floating-point number.
   - **`Unary plus opertator(+value)`**: Converts a string to a number

```
let age = '33'; Number(age); // 33 parseInt(age); // 33
parseFloat('33.5'); // 33.5 + age; // 33
```

3. ## Converting to Boolean:

```
Boolean(value): Converts a value to a boolean (true or false).
Boolean(1); // true
Boolean(0); // false
Boolean("hello"); // true
Boolean(""); // false
```

Why is Type Casting Important?

- **Handling External Data:** When working with APIs or user input, data often comes in string format, even if it represents numbers or booleans. Type casting allows you to convert this data into the appropriate type for calculations or comparisons.

- **Ensuring Data Integrity:** By explicitly converting data types, you prevent unexpected behavior caused by automatic type casting and ensure your code works as intended.

- **Working with Different Systems:** When interacting with systems that expect specific data types, type casting allows you to format your data accordingly.

Example: Handling User Input

```
let userInput = '10'; // User input is typically a string
let quantity = Number(userInput); // Convert to number for
calculations
let totalPrice = quantity * price;
```

Important:

- JavaScript performs automatic type casting, which can sometimes lead to unexpected results.
- Manual type casting gives you control over data type conversions.
- Use appropriate methods like `String()`, `Number()`, `parseInt()`, `parseFloat()`, and `Boolean()` for explicit type conversion.
- Type casting is essential for handling external data, ensuring data integrity, and working with different systems.

VIDEO:

In this guide we're going to talk about typecasting. Now that you have a good idea on what variables are how they have prospective data types, which are categories that each one of the data points fall into. A very common pattern is to have the requirement where you may need to change from one datatype into another one and that's what this guide is going to be about. To start off I'm going to go through how javascript is very forgiving and it actually tries to perform automatic type casting for you. That may seem like a good idea, until you see what that can accidentally lead to.

I'm going to start off in the console, by saying a string of 100 and I'm going to subtract 42 from that.

```
'100' - 42;
```

Now this is going to return an integer of 58. So everything worked perfectly right there. It was able to take a string, it realize that it's a string that actually a had a number inside of it. So it performed the correct computation.

Now if I did something like this where I say

```
100 + null
```

If I run this, it is `100`.

The reason for this is because javascript was able to see null, and instead of throwing an error it converted null to be zero, because it just assume that we are trying to perform some type of computation, obviously we don't want to try to take null as a value. We want to have zero in this case because it's empty.

All of that makes sense, but it starts to get a little bit tricky when we do something like this.

```
100 + 42
```

Remember up top we did `100 - 42` with 100 being a string and it all worked fine.

What do you think's going to happen now? Well if I run this you can see this returns something very different. It returns a string of `10042`.

Probably not what we were looking to do, the reason for this is because javascript doesn't know, are we just trying to append the number 42 onto the string of 100 or are we trying to perform some type of computation on it. That's something that it simply doesn't know about.

Just in case you're curious, this doesn't have anything to do with order. If I were to do

```
42 + '100'
```

It returns `42100`.

So any time that you have a string and a number and you try to add them together that is not going to work. Later on I'm going to show you how we can make it work by doing manual type casting. So, that is the automatic side of it.

Now let's talk about how we can do manual type casting. I'm going to say

```
var ageOne = 12;
```

Now, if I wanted to convert this number into a string there are two ways to do it. The first is by calling the string function here and just passing in ageOne. And this is going to return 12 but it's going to return 12 as a string.

This is where I can pass ageOne in as a function argument, which we'll talk about in a later module. But it has another syntax option. I can say `ageOne.toString` and also because this is a function I need to end it with these parentheses. If I run this you can see it performs exactly the same way.

There's really not a big difference in how this is functioning. It really comes down to a matter of what the implementation is and if it works better to have a function argument or the second one. Typically I use the two string method because typically when I'm trying to convert something, it's easier to just use that dot syntax. But I definitely recommend that you try both out. That is how you can convert numbers into strings.

Let's talk about how we can convert strings into numbers. Because there are many different ways that we can do this. I'm going to say `var ageTwo = '33';`.

The first way is very similar to how we did it with the function argument. Now if I say `number(ageTwo);` it prints out `33`.

What it's going to return is 33, but it returns it as a number. Now we do not have a `.toNumber` method.

If I tried to do something like `ageTwo.toNumber` you can see we don't get any more type of auto complete and it's because there is no such thing as a .toNumber function, but there are many other ways that we can do this.

The first that I'm going to go into after number is one called `parseInt`. Here we can pass in `ageOne` again, and that works.

I'm going to go into in a little bit exactly what parseInt does and how else we can use it.

Another option is `parseFloat`.

What this is going to do in this case is return the exact same thing.

Where this would get different is if we passed in something like this

```
parseFloat('33.5');
// 33.5
```

This will return `33.5` as a float which means a decimal point. Where as if I did something like:

```
parseInt('33.5');
// 33
```

It returns just `33`.

That's what the difference is, if you need a decimal returned then `parseFloat` works. It also works on regular integers. If you just need a regular integer and you don't care about decimals. Then you can use `parseInt` to your heart's content.

Now another spot where this comes in really handy, is with something like this. Say you have a big string not just a nice and easy number.

Let's say you have something like this

```
parseInt('5555555555555 is my phone number');
```

If I run this you can see it returns the phone number as an integer.

One thing to keep in mind is it has to start with something that can be converted to a number. If I started off with foo, just a random set of strings here and run this it's going to return NaN. Now NaN is short in javascript for not a number.

What it's telling us is it can't perform this action, it can't parse this into an integer, because the value that we pass to it is not a number. It's not something that can be converted.

The next one we're going to go into, and this is going to be the last type of conversion. This is one of my favorite ones. This is called a `unary operator` and I can say + and then just ageTwo and this converts it just like that.

```
+ ageTwo;
// 33
```

This is a very handy way of being able to do it. And I wanted to include it because in many professional javascript applications you will see this `unary operator` and it's a way of converting a string into a number. Usually you'll see it in this type of syntax, where it's:

```
var foo = + ageTwo;
// undefined
```

Then if I return this now, foo is 33.

```
foo;
// 33
```

That's the traditional syntaxes where you'll receive some type of value. You know that it's going to be a string, but it's a string that is a number and then inside of that you just use the + right in front of it, and it converts it for you. A great question to ask right now, is why in the world is this important? Because it seems like when you want to work with numbers you would simply work with a number, and when you want to work with strings you would work with the string. There are few different reasons.

One is many javascript applications are API based applications. Which means that they are communicating with the outside world and the outside world usually is going to be sending even numbers as strings because it uses a framework or a language called javascript object notation. Which we'll talk about a little bit later.

Essentially what that means is there are going to be plenty of times where you may be getting things that should be numbers back, they should be numbers they shouldn't be decimals but they're wrapped as strings and so you need to be able to parse those into strings or float. So it's something that's very common.

Another time you're going to do this (let's go back to our '100' + 42 example). Now let's imagine that we have a scenario where we received input, say from an outside application and it passed this value in. And so what we can actually do, is we can say number pass '100' in and now we have 142.

```
Number ('100') + 42;
// 142
```

Now it works! That is how you can pass those in and say you're unsure about both of them. It's perfectly fine for you to run this:

```
Number('100') + Number(42);
// 142
```

you get the same exact output.

If you're not sure what kind of data you're going to get maybe sometimes for some reason you're getting it as a string, sometimes you're getting it as an integer, then you want to be able to be confident about it. And if you put a number around it or use one of the other types of systems like

the unary operator or parseInt or parseFloat then you can be confident that your computation is going to work.

The very last thing we're going to cover in this guide, is we're going to see how we can convert booleans. If I have a boy then I can use the number function here. Pass in true, and this is going to return one. If I do the same thing to false, this is returning 0.

```
Number(true);
// 1
Number(false);
// 0
```

This is something that's very handy because in the very low level programming universe which is at the end of the day what all of our computers every server every system in the whole world uses the actual ones and zeros, it uses binary code.

All of this programming that we do has to at some point get all the way parsed down to where it's using ones and zeros. We just happen to be writing in languages we can use words and symbols in, but eventually it all has to come down to 1 or 0. This is a very handy thing, for whenever you are having to check something.

Say that you are working with a system that doesn't really have the concept of true or false and you need to be able to use a 1 or 0. Or that's what you have to return, this is how you can very quickly and easily do it.

Say that you have a function and it communicates with some other API and that API doesn't know what true or false is they can't interpret that. But it can interpret 1 and 0 and that's how it should be represented. You can wrap. True or False into it. Return it and everything works. That is how you can use typecasting in javascript.

```
"100" + 42; // "10042"
42 + "100"; // "42100"
"100" - 42; // 58
100 + null; // 100

var ageOne = 12;
String(ageOne); // '12'
ageOne.toString(); // '12'

var ageTwo = '33';
Number(ageTwo); // 33
parseInt(ageTwo); // 33
parseFloat(ageTwo); // 33
+ ageTwo; // 33
parseInt('5555555555 is my phone number'); // 5555555555
parseInt('foo 5555555555 is my phone number'); // NaN
Number("100") + 42; // 142

Number(true); // 1
Number(false); // 0
```

Coding Exercise

Give `myNumber` a value of 12 as an integer.

Then, change it to a string, saving it to `myNewString`.

```
let myNumber = //assign the integer;

let myNewString = //now change my number to a string;
```

1.10 Working with String Functions - Part 1

This part 1 guide walks through how to call functions on strings in order to perform tasks such as searching for values, finding a character's index, and much more.

THEORY:

Strings are a fundamental data type in JavaScript, used to represent text. JavaScript provides a rich set of built-in functions to manipulate and work with strings effectively.

This guide explores some of the essential string functions, empowering you to perform tasks like searching, extracting substrings, and modifying strings with ease.

Understanding Functions

Before diving into string functions, let's recap what functions are. In programming, functions are reusable blocks of code that perform specific tasks. They encapsulate behavior, allowing you to execute a series of operations with a single command. JavaScript provides many built-in functions, and you can also create your own custom functions.

String Functions in Action

Let's work with the classic pangram sentence:

```
var str = 'The quick brown fox jumped over the lazy dog';
```

1. `length` **Property:** Returns the length of the string (number of characters).

   ```
   str.length; // 44
   ```

2. `charAt(index)`**:** Returns the character at the specified index. Remember that JavaScript uses zero-based indexing, where the first character is at index 0.

   ```
   str.charAt(2); // "e"
   str.charAt(0); // "T"
   ```

3. `concat(str1, str2, ...)`**:** Combines two or more strings and returns a new string.

   ```
   str.concat(' again and again'); // "The quick brown fox jumped over the
   lazy dog again and again"
   ```

   ***Important:** String functions generally do not modify the original string.

```
They return a new string with the applied changes.
```

4. `includes(searchString):` Checks if a string contains a specified substring. Returns `true` if found, `false` otherwise.

```
str.includes('quick'); // true
str.includes('foo'); // false
```

5. `startsWith(searchString):` Checks if a string starts with a specified substring.

```
str.startsWith('The'); // true
str.startsWith('quick'); // false
```

6. `endsWith(searchString):` Checks if a string ends with a specified substring.

```
str.endsWith('dog'); // true
str.endsWith('lazy dog'); // true
```

Why Use String Functions?

- **Efficiency:** String functions provide pre-built logic for common string operations, saving you time and effort.
- **Readability:** Using string functions makes your code more concise and easier to understand.
- **Maintainability:** String functions make your code easier to update and modify.

Exploring Further

This guide covered some of the fundamental string functions. JavaScript offers many more string functions for tasks like:

- **Changing case:** `toUpperCase()`, `toLowerCase()`
- **Extracting substrings:** `substring()`, `slice()`
- **Searching and replacing:** `indexOf()`, `lastIndexOf()`, `replace()`

VIDEO:

This is going to be a little bit of a longer guide. It's because we're going to go through a number of functions related to the string data type inside a javascript. Before we go into those let me give kind of a high-level overview of what a function is because if you've never done any programming before it may be a little bit of a new concept.

Essentially a function allows you to encapsulate behavior. In this case, we're not the ones actually creating the functions, these are provided in the core javascript library, so we can simply call them. Later on, we're going to go through how we can create our own functions. But this should give you a nice little introduction because, essentially, what we're going to be able to do is to have an object and then be able to either change it or make certain kinds of value queries on it-all kinds of things.

That would take a lot of code to do if we had to write it all by hand. And the other nice thing that functions allow us to do is to perform a task again and again without having to repeat any code. That's a high-level view of what functions are. Now we're going to get into how we can use them on the string class. We'll start off by creating a variable called `str`. I'm going to set it equal to a string. We are going to say "The quick brown fox jumped over the lazy dog."

```
var str = 'The quick brown fox jumped over the lazy dog';
```

If you've ever heard that sentence before, there's a reason why it's a pretty popular one. That's because it contains every letter in the alphabet, at least one time. So it's a popular one to use for programs and also for things like font libraries this is a popular sentence you'll see. That's a little bit of a side note, just in case you've ever wondered where that kind of sentence comes from.

The first thing I'm going to do is not actually going to be a function. The first thing I'm going to do is to call what is going to be an attribute of the string. This is going to be string length. If you just add a semicolon at the end you'll see that this has forty-four characters.

```
str.length;   //44
```

Now the reason why I know this is an attribute and not a function is because if I tried to put parentheses at the end, which is what you would do if this was a function, call and hit return you will get an error. It even tells you that `.length` is not a function. That's not something that is critically important to know right now, but all of the rest of the functions I'm going to go through all have the parenthesis at the very end. I wanted to clarify why the length didn't. Running `str.length` is something you'll be doing quite a bit. It's something where you're able to check to see exactly how long a string you're working with is.

Imagine a scenario where you were rebuilding a site like Twitter. You want it to check to make sure that a certain string doesn't go over a character limit, just the same way that Twitter does it. Using string length is a quick and efficient way of doing that.

The rest are all going to be functions. It's going to follow the pattern of having parens at the very end. Sometimes we will also pass an argument to it.

The first one we're going to go with is called a `charAt` which is short for character. If I do `string.charAt` and pass in `2`, then hit return. This brings `"e"`.

```
str.charAt(2);       //"e"
```

This is going to require a little bit of explanation. If you have never worked with an index before. What we essentially did here, with this string calling `charAt` and passing this `2` in, means that we want whatever letter is there, whatever element is there, we want whatever one is at the index of two.

That seems pretty straightforward until you go up to our sentence, and you count. One is t, h is two, and e is three. But we asked for two and it returned the E. While there is a good explanation for that and that is that in computer science most of the time all of your indexes start with what's called a `zero-index` which means that the very first element is not going to be an index of one. It's an index of 0. So the next one's going to be one and the one after that is going to be two.

That's the reason why we received `"e"` back when we said character at 2 because technically this has an index of 2. It may not be the second element but from a computer science perspective that is the index of two. Now that is something that is very important to remember because not only is that going to be important when you're grabbing a character like this but this is exactly the way that when we get to work with the array data structure arrays start with a zero index as well.

If you go and you are off by one value, then this is quite possibly the reason why. You have to make sure you're always starting at zero. So that is character at, and I'm not going to spend that long with every single one of these items, it's because that was a very important topic because it relates to pretty much all kinds of counting algorithms and things like that in computer science.

The next one is still going to be `charAt` but let's see what happens if I pass in an index that is way past any existing index. I sometimes like including these because many times it's important to know because you want to have some confidence. If you call an index it doesn't exist, are you going to get an error? Or what's going to happen? Well, if I hit return you can see this simply returns an empty string. So this is good but it can also be confusing at times as well.

Sometimes I almost kind of wish that it returned null or threw some type of an error, the thing that can be confusing about this, is let's say that I did

```
str.charAt(3);      //" "
```

You can see that this may not be an empty string, it's technically a space, but it's pretty close to when we passed something that had nothing at all in there. That is something just to keep in mind if you're ever having to implement some type of coding exercise, or some program, where you may end up in a situation that you're calling an index that may not exist you will be getting an empty string back in that case. It's important to understand that it doesn't mean that it found something it just means that nothing was returned at that index value.

The next one is called `concat`. And this is short for concatenate. If I do

```
str.concat(' again and again');
```

Now you can see that it has combined what we passed as the argument to concat into and appended it to the end of our string where it's now the quick brown fox jumped over the lazy dog again and again. Now a great question to ask right now would be: have we permanently changed the value of the string? There's a pretty quick way to find that out. Just type in `str;` and you can see that it is back to the same value of when we assigned it. That's a very important thing to know right there because if you run concat thinking that you're changing the string, you really aren't, you're simply changing the value that got returned from running that `concat` function on it.

If you wanted to permanently connect those two then you would want to do something like

```
var newStr = str.concat(' again and again');
```

And that puts that in a new string, your new string now contains those values. But the preexisting string doesn't, so that is a very important thing and that is the case for every one of these string functions that we're going to cover. That was Concat.

Now let's go into some matches. This is going to give us the ability to see if any values are included in the string. This is something that's very common to do. Say you're working with a string and you want to see if it contains this other value I'm looking for.

If I run `str.includes();` and pass in quick. This is going to simply return true.

```
str.includes('quick');   // true
```

It goes through our string, and it found quick right there, and it said, yes, it does include that. Now if I were to run `str.includes();` again and pass in foo, which does not exist, it returns false. So includes is a very nice and straight forward type of function where you pass in whatever you're looking for. If it includes that value, it returns true. If not, it returns false.

Now there are some more customized ways of doing this. We have included, but we also have a function called `startsWith`. `startsWith` is very similar except instead of looking through the whole string. It just looks at the very front. If I pass in "the" it returns true because it starts with that.

If I pass in quick it returns false because even though that's in the string it doesn't start the string out.

Now, this has kind of a contrapositive here which is `endsWith`. We have `startsWith` and `endsWith`. When it's quick, it's going to be false. If it is dog it's going to be true, which you probably already figured out. But what if it's just "g" that's also true. It doesn't check for words it checks just character by character. If I were to say `str.endsWith('lazy dog');` it is also true.

That's an important thing, because if you're just looking at it, you're looking at all these words, you may think "Oh, it's looking for the last word and I need to have the last word to put it in there". But you may just be looking through some raw data files and you want to find out which ones end in a semi-colon, or anything like that. This is a nice quick way of doing that. I recommend you to go through these again. Play with them, see how you can change them to manipulate the text.

```
var str = 'The quick brown fox jumped over the lazy dog';

str.length(); // VM2349:1 Uncaught TypeError: str.length is not a
function

str.length; // 44

str.charAt(2); // "e"

str.charAt(200); // ""

str.concat(' again and again'); // "The quick brown fox jumped over
the lazy dog again and again"

str; // "The quick brown fox jumped over the lazy dog"

str.includes('quick'); // true

str.endsWith('dog'); // true

str.startsWith('Foo'); // false
```

Coding Exercise

Call these two functions on the provided string and have them return true. The two functions are endsWith and startsWith.

```
string = "Hello, what happened to all the pie?"
```

1.11 Working with String Functions - Part 2

In this second part of our module on String functions in JavaScript, we'll examine how to integrate Regular Expressions in order to find patterns in string based data.

THEORY:

Recap from Part 1:

We're working with the following string:

```
var str = 'The quick brown fox jumped over the lazy dog';
```

String Functions with Regular Expressions

1. `repeat(count)`: Repeats a string a specified number of times.

```
str.repeat(5);
// "The quick brown fox jumped over the lazy dogThe quick brown
fox jumped over the lazy dogThe quick brown fox jumped over the
lazy dogThe quick brown fox jumped over the lazy dogThe quick
brown fox jumped over the lazy dog"
```

Important: Like other string functions, repeat() does not modify the original string; it returns a new string with the repeated value.

2. `match(regexp)`: Searches a string for a match against a regular expression and returns an array of matches.

```
// Matching a phone number pattern
let phoneNumberPattern = /((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}/;
str.match(phoneNumberPattern); // null (no match in the original
string)
'555-555-5555'.match(phoneNumberPattern); // Returns an array
with the matched phone number
```

Regular expressions provide a concise and flexible way to define patterns. You can find pre-built regular expressions for common patterns online or learn to create your own.

3. `replace(searchValue, replaceValue)`: Replaces occurrences of a specified value within a string. You can use a string or a regular expression as the `searchValue`.

```
str.replace('fox', 'wolf'); // "The quick brown wolf jumped over
the lazy dog"
// Replacing multiple spaces with a single space
let strWithExtraSpaces = "This  has   extra    spaces";
strWithExtraSpaces.replace(/\s+/g, ' '); // "This has extra
spaces"
```

4. `search(regexp)`: Searches a string for a match against a regular expression and returns the index of the first match. Returns -1 if no match is found.

```
'555-555-5555 is my phone number'.search(phoneNumberPattern); //
0 (match found at index 0)
'Hi, 555-555-5555 is my phone
number'.search(phoneNumberPattern); // 4 (match found at index
4)
str.search(/foo/); // -1 (no match found)
```

5. `indexOf(searchValue)`: Returns the index of the first occurrence of a specified value within a string.

```
str.indexOf('jumped'); // 20
```

6. `lastIndexOf(searchValue)`: Returns the index of the last occurrence of a specified value within a string.

```
let str2 = str.concat(' again and again');
str2.indexOf('again'); // 44 (first occurrence)
str2.lastIndexOf('again'); // 54 (last occurrence)
```

Why Use Regular Expressions with String Functions?

- **Complex Pattern Matching:** Regular expressions excel at finding and manipulating complex patterns in strings that would be difficult to achieve with basic string functions alone.

- **Data Validation:** Regular expressions are commonly used for validating user input, such as email addresses, phone numbers, and passwords.

- **Text Manipulation:** Regular expressions can be used for advanced text manipulation tasks like extracting specific data from strings or reformatting text.

VIDEO:

Hi, and welcome back! As we go through the full list of string functions in Javascript. Let's bring back our variable.

```
var str = 'The quick brown fox jumped over the lazy dog';
```

With that in place let's keep going.

The next one on the list is repeat. If I do str.repeat then as an argument in the parens I can pass in however many times I want to repeat that line.

```
str.repeat(5);     // "The quick brown fox jumped over the lazy dogThe
quick brown fox jumped over the lazy dogThe quick brown fox jumped
over the lazy dogThe quick brown fox jumped over the lazy dogThe quick
brown fox jumped over the lazy dog"
```

Now you can see that this is printed out. The quick brown fox jumped over the lazy dog five times and it put it all in one string. Now once again just to reinforce it, this did not alter the original string. This simply returned a value. And this really goes to the heart of how functions work in javascript. It's pretty rare that you want to actually alter the variable. And in fact, the with the way that variables work in this gets into a little bit more of an advanced topic that we'll get into later. The variables in javascript when they get passed to functions they simply pass the value. They don't pass in a reference to the variable. In other words, when something changes that or alters it, it doesn't go back and change the original variable. It simply returns an updated set of values. So in this case when we ran repeat five times it only gave us that value back.

We could use that we could store it in another variable we could wire that up into another function. Anything like that and that would all work. But we still can be confident that our variable string didn't get changed.

Next on the list is a little bit of a different one. If you've never heard of regular expressions, this is going to look very weird. A regular expression is a pattern matching system. This is pretty much available to every programming language at least all of the popular general-purpose programming languages. Javascript, Ruby, Perl, Java, any of these languages they have a reference to what is called a regular expression and the nice thing is, it's actually pretty much the same across all those languages. Now because of that regular expressions look very weird. The next function we're going to do is called `match`. This is going to take in a regular expression and then it's going to tell you if it was a match. I want to build a regular expression matcher for phone numbers.

I want to know if a string contains a phone number. I'm going to go off-screen because you do not want to watch me type this out and I'm going to grab this regular expression and paste it in.

```
str.match(/((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}/);
```

You can see why you wouldn't want to watch me type this one, I would probably type one little thing off and then it wouldn't work, and it would take a while. But essentially what this says is this is a pattern matching system. Now don't let this intimidate you, and don't think that you're going to have to be able to write these from scratch.

That definitely falls into the vein of very advanced so much so that I'd say half the regular expressions that I use. Usually, I just Google it exactly like I did this one. I googled regular expression for a phone number and I was greeted with hundreds of people who had put in their own patterns. So essentially through what this is going to do is this is going to be compared with javascript or javascript's going to use this to compare it with our string and it's going to say does this pattern match what's contained in the string. Now there is no phone number here, so this should

just return null. If I run it, it does, it doesn't error out. all this essentially means is that there is no phone number and there's nothing that is recognized as a phone number.

Now if I change this and I'm not going to use a variable I can just use a regular string and I change this to be (5 5 5) 5 5 5 - 5 5 5 5 just like a phone number pattern if I hit return. Now, look at that. That gives us a full object back.

Now don't worry about knowing every one of these items, because some of these we haven't gone over yet such as arrays and nested objects inside of arrays. But if you click on this you can see that it did in fact find this pattern. If you really want to spend some time understanding regular expressions a little bit more if you come in, you can see that right here it has things like the parens. It's looking for the parens right here.

Then this little `d{3}` inside of curly brackets means it's looking for three numbers inside of those curly brackets and then it's going to do the same thing here with a dash followed by it and then another one right here where it's going to be four numbers. That's what it is. But there are entire courses dedicated to regular expressions. You definitely don't have to worry about that right now. But what I want to get across is that you have the ability to use the match function in order to see if something that you're wanting to check against actually is a match or not. This works very well and is required for things such as web form validations.

Imagine that you're building out a web application and you have an email field which is a pretty common thing to have. Say that you want to make sure that someone can't just type in any random set of characters and expect that to be able to be submitted in the form for an e-mail filled. You want to be able to make sure that what they type in matches what an e-mail is. What you could do is build in a match or find an email regular expression pattern and then say does what they typed in does that match? If so let it go on, If not tell them that they have to enter in a valid email. That's a very common reason why you'd want to use the match function.

Next on the list is `replace`. Say that you want to replace something. this takes into arguments. The first is what you're searching for. So here you can say Fox and the second argument separated by a comma is what you want to replace it with. So if I want to replace Fox with Wolf, I would run this. Now you can see it says the quick brown Wolf jumped over the lazy dog.

```
str.replace('fox', 'wolf');    // "the quick brown wolf jumped over
the lazy dog"
```

And you can also if you wanted to you could pass in a regular expression as that first argument because there are many times where you don't know what you're looking for. So in other words you're not looking for a specific word. You may be looking for a pattern and then you want to replace it with some other value. So that's also a very helpful thing to know, that is string replace.

The next one is going to be search. This one I'm going to copy again because I thought the example that worked best was a phone number one again. I'm going to paste this in then we'll talk about it.

Here we have a valid phone number. And also notice this pattern right here doesn't require the parentheses that also allows for this kind of dash notation right here. Here it's saying this is my phone number search and then we're passing in a regular expression. And if I run this it says zero. So what exactly is search doing? Well, what search does is, it looks and when it finds a pattern that it matches with, then it will actually return the index. So zero does not mean that it didn't find it. It means that it did find it and it found it at the zeroth index. Now if I run this again with just `foo`. I run this and it returns negative one.

Now a negative one means it didn't find anything. So if you're running a search and you're going to be checking to see is this inside of it? If it is, it's going to give you an index starting at zero going all the way through the length of it. And if it's not then it will give you a negative one. I'll also note that it's very similar among other programming languages. Ruby does a very similar kind of implementation for regular expression searching and matching as well.

Now instead of that let's just check this out just so you know that I'm not lying about this zero. I say foo here and run it, now it returns 4. That's because if we count, remember this is the 0 index. So, zero, one, two, three, and then four. So what it found, the pattern that it found starts on index four and that's the way that `search` works.

Now that we're kind of talking about indexes let's go into it in more detail with `indexOf`. If I look at our string again I'm going to pull that up so we can reference it. Now we can pass in a word so I can say jumped and run this.

```
str.indexOf('jumped');     // 20
```

You can see that it is at index 20. And now one thing to keep in mind is that this is index 20 but it is the index of the first word that it finds. Now we only have one jumped word, but let me create a new string here. I'm going to create var string 2 and we'll get to practice our concating here. So you say str.concat and pass in what we did before "again and again".

```
var str2 = str.concat(' again and again');
```

But now we have this new string too that actually has the full string plus what we put at the end of it. So now what we can do which is pretty cool is, if we say

```
str2.indexOf('again');    // 45
```

and run that you can see it's 45. But now we can also do another function that's pretty close to it which is `lastIndexOf`. And now you can see it's 55.

```
str2.lastIndexOf('again');     //  55
```

What the difference there is, `indexOf` looks through the whole string, once it finds the very first instance of what it's searching for it just turns that index, last index will go through and it goes through every one it finds every one of the patterns or the words we're looking for. And then it returns the last one that it found.

That is something that's very helpful. And if you ever need to find all of them that's what going back to the match function, that's what the match function does. So that is quite a bit on how to use regular expressions. I know this is another guide that took a while and so I'm going to stop it right now and then we're going to finish off with a part 3. That is going to finish off the rest of the string functions.

```
str.repeat(5); // "The quick brown fox jumped over the lazy dogThe
quick brown fox jumped over the lazy dogThe quick brown fox jumped
over the lazy dogThe quick brown fox jumped over the lazy dogThe quick
brown fox jumped over the lazy dog"

str.match(/((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}/) // null

'555-555-5555'.match(/((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}/) // (4)
["555-555-5555", "555-", undefined, "555-", index: 0, input: "555-555-
5555"]0: "555-555-5555"1: "555-"2: undefined3: "555-"index: 0input:
"555-555-5555"length: 4__proto__: Array(0)

str.replace('fox', 'wolf'); // "The quick brown wolf jumped over the
lazy dog"

'555-555-5555 is my phone number'.search(/((\(\d{3}\) ?)|(\d{3}-))?
\d{3}-\d{4}/) // 0
```

```
'Hi, 555-555-5555 is my phone number'.search(/((\(\d{3}\) ?)|
(\d{3}-))?\d{3}-\d{4}/) // 4

str.indexOf('jumped'); // 20

str.lastIndexOf('jumped'); // 20

var str2 = str.concat('again and again');

str2.indexOf('again'); // 44
str2.lastIndexOf('again'); // 54
```

Coding Exercise

For this submission, replace dog with cat, find the index of over, and find the last index of never.

```
stringOne = "The dog meows"
replacedString =  // replace the word dog with cat

stringTwo = "The cow jumped over the moon"
indexOfOver =  // get the index of over

stringThree = "Never gonna give you up never gonna let you down"
lastIndex = // get the last index of never
```

1.12 Working with String Functions - Part 3

This is the final guide in the String function module. In this lesson we'll walk through the slice, case, and trim functions. We'll also examine how we can chain functions together.

THEORY:

Recap from Paets 1 & 2

We're continuing to work with the following string:

```
var str = 'The quick brown fox jumped over the lazy dog';
```

More String Functions

1. `slice(startIndex, endIndex)`: Extracts a section of a string and returns it as a new string.
    - **`str.slice(startIndex)`**: Returns a substring from `startIndex` to the end of the string.
    - **`str.slice(startIndex, endIndex)`:** Returns a substring from `startIndex` up to (but not including) `endIndex`.
    - You can use negative indexes to slice from the end of the string.

```
str.slice(10); // "brown fox jumped over the lazy dog"
str.slice(-8); // "lazy dog" str.slice(4, 9); // "quick"
```

2. `trim()`: Removes whitespace from both ends of a string.

```
var messyString = '    Hi there    ';
messyString.trim(); // "Hi there"
```

3. **Chaining String Functions:** You can chain multiple string functions together for more complex operations.

```
str.slice(4, 10).trim(); // "quick" (extracts "quick " and then
removes the trailing space)
```

4. `toUpperCase()`: Converts a string to uppercase.

```
str.toUpperCase(); // "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"
```

5. `toLowerCase()`: Converts a string to lowercase.

```
str.toLowerCase(); // "the quick brown fox jumped over the lazy dog"
```

Why These Functions Matter

- **Data Cleaning:** `trim()` is invaluable for cleaning up user input or data from external sources.
- **String Manipulation:** `slice()` provides precise control over extracting portions of strings.
- **Case Conversion:** `toUpperCase()` and `toLowerCase()` are useful for string comparisons and formatting.
- **Code Efficiency:** Chaining functions can make your code more concise and readable (when used judiciously).

Beyond the Basics

This series has covered the core string functions in JavaScript. There are other more specialized functions available, such as:

- `substring()`: Similar to `slice()`, but handles negative indexes differently.
- `split()`: Divides a string into an array of substrings based on a separator.
- `localeCompare()`: Compares strings according to language-specific rules.

VIDEO:

Welcome back to this third and final video on how to work with string functions in javascript. This one should also be a little bit shorter we just have a few items that I wanted to cover.

The first one is the `slice` function so I'm going to type string or str because that's the name of my variable dot slice. Now we have a few different options. So we have three ways that we can use slice, the very first way is to just pass an index value and what it will do is it will return all of the items to the right of that.

```
str.slice(10);
"brown fox jumped over the lazy dog"
```

So let's just do

```
str.charAt(10);
"b"
```

to see where that's at. That is the b, so it grabs the b all the way to the end. That is a nice and easy way if you know that you have some values that you want to skip over and you don't care about them and you just want from that point all the way the end. `Slice` with just one value will give you that. And it's that index value.

Now the other way we could do this is we could actually go backwards. So say that we have a giant string and we don't want to count all the way from left to right. And also another time where that

comes in very handy is say we have a set of strings. So it could be a social security number or something like that. And we know that we're always going to get say the last four or five digits. Those are always going to be exactly the same. And we only want those.

Well, we can work from right to left by using negatives. So here I'm going to say negative 8 and run this.

```
str.slice(-8);
"lazy dog"
```

You can see that returns the lazy dog. So far those are two different options one of them gives you all of the items and you're counting from left to right. If you use negatives you're counting backwards. But at the end of the day, you're essentially still finding an index and it's returning everything from that index all the way to the right. Even if you're counting from left to right or right to left.

The last option is to actually grab a slice inside of the string. So let's say that I only wanted the word quick here and I knew that this was an index 4 through 10. The way that I could do this is actually passing two arguments. So I could do `str.slice` and if I want quick I can do four because counting from left to right this is 0 1 2 3 4. And then if I want 10 let's count 0 1 2 3 4 5 6 7 8 9 10. So if I run this you can see it returns quick and it has the little space after it.

```
str.slice(4, 10);
"quick "
```

Now we can refine that a little bit and we can say four to nine, run that. And it's quick just by itself.

```
str.slice(4, 9);
"quick"
```

Now, this also brings up something that is a little bit handy.

The next function I want to talk about is the `trim` method. So say that I have a new variable I'm going to call it messy string and this has a bunch of empty space in between. It has this foo value and then to the left has all these spaces and then to the right, it has all these spaces.

```
var messyString = '    foo     ';
undefined
```

If I call messyString you can see that it keeps all those spaces.

```
messyString;
"    foo    "
```

But if I call messyString and then call `trim` on it, this `trim` function is going to return `"foo"`. So trim is a very helpful function. You will find it is incredibly beneficial when you're wanting to clean up user data because there are all kinds of times where I'll be given a database or a raw file and it has strings left and right and trim makes it very nice and easy to clean those. So that's helpful.

Now let's see how we can actually combine some of these functions. Now remember with string when I called slice on it and past 4 thru 10 remember how that brings, quick with the extra space at the end? Well, I can actually chain these together so if I say trim and run this.

```
str.slice(4, 10).trim();
"quick"
```

Notice how this has now cleaned it up even though we're still passing four and ten.

One of the most important things I wanted to show you right there, is that these functions can be chained together. And as you go further on your javascript development journey you are going to discover that you're going to be changing methods left and right now you have to be very careful about that. There are many times where people kind of get out of hand with this and it becomes very confusing and very difficult to debug.

There are some recommended principles that say you don't really want to ever have more than two to three different items that are chained together. Anything above that it can lead to some very challenging bugs to get fixed and it's also very hard to refactor that code and clean it up. But there are going to be times where you want to do things just like this, where we `slice` something and then we called `trim` to make sure that we had some nice clean values afterwards. So that is the `slice` and then the `trim` function.

We just have two more that we're going to cover.

The first is going to be `toUpperCase` so when you call this it's going to do exactly what you thought. It's going to convert all of the values to uppercase.

```
str.toUpperCase();
"THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"
```

Now we also have an option of doing `toLowerCase` running that you can see it's all lower case.

```
str.toLowerCase();
"the quick brown fox jumped over the lazy dog"
```

Originally if you reference the very top line we only had one item that was uppercase and so that one got taken down to the lower case and all the other ones simply remain the same.

In the last three videos, we've covered all of the main functions related to the string data types. Very nice job going through that. I'm also going to include in this guide a link to the documentation that shows all of the remaining functions.

Now I didn't include every single one because that would have taken even longer and we spent quite a bit of time and I want to start moving on to other things like the number data type but also some of the ones I left out were related to things such as converting to a specific international locale to do things like that. And I know that doesn't apply to everybody. So you feel free to reference the documentation. And as always I will put a link to the source code and then also the actual code that we just went through in the written guide.

```
str.slice(4, 10); // "quick "

str.slice(-8); // "lazy dog"

str.slice(10); // "brown fox jumped over the lazy dog"

str.toUpperCase(); // "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"

str.toLowerCase(); // "the quick brown fox jumped over the lazy dog"

var messyString = '    Hi there    ';

messyString.trim(); // "Hi there"
```

Coding Exercise

Fill in the return statement below with some of the string function syntax you just learned!

```
//We have given you the sentences already, please fill in the return
statement to satisfy the requirement
// EXAMPLE "return (str.toUpperCase());"

function strings() {

    // Use slice to return a substring starting from the "w" through
the end of the sentence
```

```
    var str = "The five boxing wizards jump quickly";
    return (  <--Delete-this-and-write-your-string-function-->  );

}

strings();
```

1.13 JavaScript Arithmetic Operators

This guide examines the full list of JavaScript arithmetic operators. These operators allow for programs to perform computation and can update numbers.

THEORY:

Arithmetic operators are fundamental building blocks in JavaScript, allowing you to perform mathematical calculations and manipulate numerical values. This guide provides a detailed overview of JavaScript's arithmetic operators, ranging from basic operations to more advanced techniques.

Basic Arithmetic Operators

JavaScript supports the standard arithmetic operators:

- **Addition (+):** Adds two numbers.

```
2 + 2; // 4
```

- **Subtraction (-):** Subtracts one number from another.

```
2 - 3; // -1
```

- **Division (/):** Divides one number by another.

```
10 / 2; // 5
```

- **Multiplication (*):** Multiplies two numbers.

```
2 * 10; // 20
```

- **Exponentiation ()**:** Raises a number to the power of another number.

```
2 ** 10; // 1024
```

- **Modulus/Remainder (%):** Returns the remainder left over when one number is divided by another.

```
5 % 2; // 1

Practical Use of Modulus: The modulus operator is often used to
determine if a number is even or odd. A number is even if its
remainder when divided by 2 is 0.

10 % 2; // 0 (even)
7 % 2;  // 1 (odd)
```

Increment and Decrement Operators

- **Increment (++)**: Increases the value of a variable by 1.
  - **Postfix (`num++`)**: Returns the original value before incrementing.
  - **Prefix (`++num`)**: Returns the incremented value.

```
let num = 2;
num++; // 2 (num is now 3)
++num; // 4 (num is now 4)
```

- **Decrement (--)**: Decreases the value of a variable by 1.
  - **Postfix (`num--`)**: Returns the original value before decrementing.
  - **Prefix (`--num`)**: Returns the decremented value.

```
let num = 4;
num--; // 4 (num is now 3)
--num; // 2 (num is now 2)
```

Other Arithmetic Operators

- **Unary Negation (-):** Changes the sign of a number.

```
let num = 10;
let negNum = -num; // -10
```

- **Unary Plus (+):** Can be used for explicit number conversion (type casting).

```
let strNum = '100';
let num = +strNum; // 100 (number type)
```

VIDEO:

In this lesson, we're transitioning and we're starting to work with the number data type in javascript and specifically, we're going to analyze the various arithmetic operators available so that we can perform computation.

Now some of these are very basic, but later on, we are going to get into some more advanced kinds of operators. We're going to start off with the very basic ones and I'm going to keep the numbers very simple just so it all makes sense.

```
2 + 2; // 4
```

Not too surprisingly that equals four. And that is the syntax. Now you can also wrap these in parentheses and they'll work exactly the same way. And this will mean much more when we start to talk about the order of operations. And when we want certain computations to be performed versus other ones but we'll save that for another guide.

The next one we're going to do is just

```
2 - 3; // -1
```

That will give us -1 just to show that you also have the ability to work with negative numbers.

Now if I go

```
10 / 2; // 5
```

So far none of this is crazy. It's more of just kind of walking through the syntax.

Now for multiplication. We use the asterisks and we use a single asterisk.

```
2 * 10; // 20
```

And the reason why we use a single Asterix is because when you want to use exponents you're going to use two asterisks.

If I do

```
2 ** 10; // 1024
```

That is how you can use exponents.

Now the next thing that I'm going to walk there may seem kind of weird because there actually is what's called a modulus or a remainder operator in Javascript. And so that will allow us to do, something like this

```
5 % 2; // 1
```

This gives us a remainder of 1 which is accurate.

Now if you are like many students when you see that OPERATOR. The first thing that may pop in your head is I can never imagine a time where I will need to know what the remainder of something is. And technically that seems logical. However, I'm going to show you a very practical reason why this is a very powerful type of tool when used properly.

Imagine that you have a list of numbers and say that you have a table and you want to show every other record. Think of something kind of like Excel so say you're building a table that shows on a web application and you want every other table element. So either the even elements or the odd elements you want them to have a different background color. Well, how are you going to figure out? Which records are even and which ones are odd. Well technically if you use the modulus operator properly you will always know because if you do something like say `10 % 2` this is going to equal 0. Now let's look at `22222 % 2` that is also 0. If you do `6 % 2` that is also 0.

```
10 % 2;
0
22222 % 2;
0
6 % 2;
0
```

This is where you see the modulus operator used the very most, whenever you need to find if a number is even or odd because if it is even it is always going to have a remainder of zero when it's some number % 2. So that is something that is very handy and you actually see this in quite a few programming exercises. So that's a good thing to keep in mind and that's when you would want to use it again. Those are the main operators.

Now we're going to get into some of the ones that are a little bit different. Maybe you haven't seen them before or haven't really thought about using them. I'm going to create a variable called var num. Let me set it equal to 2. And now I'm going to use what's called the `increment` operator. So here I can say num++; and this is going to increment. But there's a little bit of a twist on this. So if I hit return you can see it returns 2.

```
var num =2;
undefined
num++;
2
```

Whatever gets shown in the console and this is why I've been using the console lately instead of codepen. And you're perfectly fine using CodePen but when you do that you have to console log everything out and that takes longer. Right here I'd prefer for you to immediately see what gets returned. This means that you put this in a function or you call this from some other place. This is the number that gets returned which is 2.

But that doesn't make any sense because we started at two. Now I'm going to show you something that looks really weird. If I just print out the number again and hit return it's three.

```
num;
3
```

Nothing happened here. It didn't change to three here. As soon as I called `Increment` this changed it. but it returns the preexisting value. So whenever you use the incrementor you put this in what's called the postfix notation postfix meaning that the operator is placed at the end. Then the number that gets returned is the first number of the preexisting number. It still does its job it still increments it. But this is something where say, that you want to keep a copy of the variable. You could have a copy of the old pre-change pre incremented version. You could store that and then your new number has been changed. So that's one example of when you'd want to use it but now say that's not very intuitive. You don't really like that. Well, you have what's called a prefix notation and that is where you put the operators upfront.

```
++num;
4
```

Now I can call num and look at that. Now it's four. So it was three here and now it's four but it also returns four. I usually find myself using the prefix notation because most of the time I just forget that the postfix notation changes it. It does its increment but then it returns the old value. I've only had a few times where I really needed that type of behavior. Usually, the prefix notation is the type of syntax I use but it's good to know that they're both there.

Now that we have four let's use the `Decrementer` operator so I can do num and then two minus signs and this is going to have the identical behavior. What you think is going to happen? In viewing all of this in viewing everything that's happening here what is going to happen with this `decrement` operator. Think about it. I'm going to hit return and see what happens. It looks like nothing happened. Well let's test that hypothesis, if I type num in again we can see it did change it. It decreases it by one but because we use the postfix notation it worked exactly the same as when we did the `Incrementor`. Now if we want to do something like we did before and actually had the return value mimic what we're wanting we can do --num; And now it takes it down to two which is the value. So it `decrement`ed it and it gave us the value that we expected as the returned value.

```
--num;
2
```

Now we don't want you to get too confused with understanding what it means for something to be returned. We're going to go into depth on what that is and how to work with return values and really what that entire workflow is when we get to the module on functions. So that's when we're going to do that for right now just know that the value that gets returned is a value we can actually work with. We can store it in a variable we could put it in a function we could do anything we wanted to it. But for right now just realize that that is a value that comes back to us after we've run whatever process we're running.

Now there's also a little caveat to the `incrementor` and the `decrementer` and that is, that you have to call it on a variable. Watch what happens if I did this 2++;

that it gives us an error of uncaught reference error invalid left-hand side expression in postfix operation.

Now if you think about this, that makes sense because if you ever had an actual hard-coded value you had a number, not a variable that stored a number. We actually had a number. Why would you want to use the increment or the decorator? Because if you actually knew the number and you were the one typing it in the program wasn't generating it, it wasn't dynamic because it wasn't in a variable. If you wanted one more than two you could just type three in.

That's just a little caveat a little side note so that you can kind of think through that, there is a difference between variables and the numbers that they store. Remember variables are special containers that can hold values. Now that we have all of that, we've gone through traditional operators, we've gone through `incrementors` and `decrementers`. We only have a few more functions we're going to go through. The next one is going to be a way that we can actually flip the values so we can get a negative on this value. I'm going to call var and then someNum = 10.

And now if I want to assign this but I want the opposite. In this case, I want negative 10. Then I can do this I can say var someOtherNum and then just do minus and then some numb. And now if I call. Some other Num. this gives us negative 10.

```
var someNum = 10;
undefined
var someOtherNum = -someNum;
undefined
someOtherNum;
-10
```

Now that may be what you expected but I just wanted to show you that, that was possible. Now we've already referenced this one when we were talking about converting different strings into numbers. But I wanted to cover the `unary` operator one more time, just because it is something that is very powerful and you're going to see it quite a bit in code. It's good to have a good idea of it.

I'm going to say

```
var strNum = '100';
undefined
```

Now if I want to convert it, I can say

```
var convertedNum = + strNum;
undefined
```

Now if I just want to see it.

```
convertedNum;
100
```

You can see that it is no longer a string. It is now part of the number data type. Also, you didn't have to put it in a variable, that's just the common convention you're usually going to see because it's pretty rare that you just have code, for example, say that we just had `+ strNum;` This would work but I wanted to show you the kind of syntax is usually going to see out in the wild. Typically when you're converting different data types usually are going to be putting them in a variable to be used in a function or something like that.

In this guide just as a quick review, we covered all of the basic arithmetic operators from Plus(+) all the way through the modulus(%) operator. We talked about `incrementors` and `decrementers` which allow us to increase or decrease values by 1 and then we also talked about how we could flip values. And we finished off with seeing how we can convert a string-based number into a number.

```
2 + 2; // 4
```

```
2 - 2; // 0

2 / 2; // 1

2 * 10; // 20

5 % 2; // 1

10 % 2; // 0

8 % 2; // 0

2 ** 10; // 1024

var num = 2;

num++; // 2

++num; // 4

num; // 4

num--; // 4

num; // 3

--num; // 2

2++; // VM3506:1 Uncaught ReferenceError: Invalid left-hand side
expression in postfix operation

++2; // VM3508:1 Uncaught ReferenceError: Invalid left-hand side
expression in prefix operation

var someNum = 10;

var someOtherNum = -someNum;

someOtherNum; // -10

var strNum = '100';

var convertedNum = + strNum;

convertedNum; // 100
```

Coding Exercise

Add the correct arithmetic operators to make variable equal the number in the comment.

```
numOne = 5 10 // add the right Arithmetic Operators to have it equal
15

numTwo = 90 3 // add the right Arithmetic Operators to have it equal
30

numThree = 50 25 // add the right Arithmetic Operators to have it
equal 25

numFour =  20 5 // add the right Arithmetic Operators to have it equal
100
```

1.14 Guide to Compound Assignment Operators in JavaScript

Compound assignment operators provide an efficient and readable way to update variables by combining arithmetic operations with assignment

THEORY:

What is Assignment?

In JavaScript, the assignment operator (=) assigns the value on the right-hand side to the variable on the left-hand side.

```
let name = 'Tiffany';
```

Compound Assignment Operators

Compound assignment operators combine an arithmetic operator with the assignment operator to perform an operation and assign the result in one step.

```
let sum = 0;
let gradeOne = 100;
let gradeTwo = 80;

sum += gradeOne; // Equivalent to: sum = sum + gradeOne; (sum is now
100)
sum += gradeTwo; // Equivalent to: sum = sum + gradeTwo; (sum is now
180)
```

Common Compound Assignment Operators

- += (Addition assignment)
- -= (Subtraction assignment)
- *= (Multiplication assignment)
- /= (Division assignment)
- %= (Modulus assignment)

Benefits

- **Conciseness:** Reduces code and makes it more compact.
- **Readability:** Improves code readability, especially with complex calculations.

Example: Updating a Value

```
let count = 10;
count += 5;  // count is now 15
count -= 3;  // count is now 12
count *= 2;  // count is now 24
```

VIDEO:

Now that we've talked about operators. Let's talk about something called the `compound assignment` operator and I'm going make one little change here in case you're wondering if you ever want to have your console take up the entire window you come up to the top right-hand side here you can undock it into a separate window and you can see that it takes up the entire window.

So just a little bit more room now.

Additionally, I have one other thing I'm going to show you in the show notes. I'm going to give you access to this entire set of assignment operators but we'll go through a few examples here. I'm going to use the entire window just to make it a little bit easier to see.

Let's talk about what `assignment` is. Now we've been using assignment ever since we started writing javascript code. You're probably pretty used to it. Assignment is saying something like var name and then setting up a name

```
var name = 'Tiffany';
```

And that is assignment the equals represents assignment.

Now javascript gives us the ability to have the regular assignment but also to have that assignment perform tasks. So for example say that you want to add items up so say that we want to add up a total set of grades to see the total number of scores. I can say var sum and assign it equal to zero.

```
var sum = 0;
```

And now let's create some grades.

```
I'm going to say var gradeOne = 100.
var gradeOne = 100;
```

and then var gradeTwo = 80.

```
var gradeTwo = 80;
Now with both of these items in place say that we wanted to add these
if you wanted to just add both of them together you definitely could
do something like sum = (gradeOne + gradeTwo); and that would work.
However, one thing I want to show you is, there are many times where
you don't have gradeOne or gradeTwo in a variable. You may have those
stored in a database and then you're going to loop through that full
set of records. And so you need to be able to add them on the fly. And
so that's what a compound assignment operator can do.
```

Let's use one of the more basic ones which is to have the addition assignment.

```
sum += gradeOne; // 100
```

Now you can see that sum is equal to 100.

Then if I do

```
sum += gradeTwo; // 180
```

If we had 100 grades we could simply add them just like that.

Essentially what this is equal to is it's a shorthand for saying something like

sum = sum + whatever the next one is say, that we had a gradeThree, it would be the same as doing that. So it's performing assignment, but it also is performing an operation. That's the reason why it's called a compound assignment operator.

Now in addition to having the ability to sum items up, you could also do the same thing with the other operators. In fact literally, every one of the operators that we just went through you can use those in order to do this compound assignment. Say that you wanted to do multiplication you could do sum astrix equals and then gradeTwo and now you can see it equals fourteen thousand four hundred.

```
sum *= gradeTwo; // 14400
```

This is that was the exact same as doing sum = whatever the value of sum was times gradeTwo. That gives you the exact same type of process so that is how you can use the compound assignment operators. And if you reference the guide that is included in the show notes. You can see that we have them for each one of these from regular equals all the way through using exponents.

Then for right now don't worry about the bottom items. These are getting into much more advanced kinds of fields like `bitwise` operators and right and left shift assignments. So everything you need to focus on is actually right at the top for how we're going to be doing this. This is something that you will see in a javascript code. I wanted to include it, so when you see it you're not curious about exactly what's happening.

It's a great shorthand syntax for whenever you want to do assignment but also perform an operation at the same time.

```
var name = 'Tiffany';

var sum = 0;

var gradeOne = 100;

var gradeTwo = 80;

sum += gradeOne; // 100

sum; // 100

sum += gradeTwo; // 180

sum *= gradeTwo; // 14400
```

Coding Exercise

You need 250 lemons and 36 limes for your lemonade. Whats your total number of fruit?

```
//Use Compound Assignment Operators to solve the above problem

function mathTest() {
    //please do not delete this
```

```
    var sum = 0;

    //Delete this and assign your variables, then do some math

    //please do not delete this
    return sum;
}

mathTest();
```

1.15 Order of Operations in JavaScript: Understanding PEDMAS/PEMDAS

JavaScript follows the PEDMAS/PEMDAS order of operations.
Multiplication and division, as well as addition and subtraction, have equal precedence and are evaluated from left to right.
Using parentheses to control the order of operations in order toavoid ambiguity.

THEORY:

Just like in mathematics, JavaScript follows a specific order of operations when evaluating expressions. This order ensures that calculations are performed consistently and predictably. This guide explains the order of operations in JavaScript, including a discussion on PEDMAS/PEMDAS.

Example

Consider the following expression:

```
let num = 5 + 5 * 10;
console.log(num); // Output: 55
```

You might expect the output to be 100 (5 + 5 = 10, then 10 * 10 = 100) if you evaluate from left to right. However, JavaScript prioritizes multiplication over addition, resulting in 55 (5 * 10 = 50, then 50 + 5 = 55).

PEDMAS/PEMDAS

The order of operations in JavaScript is often remembered by the acronyms PEDMAS or PEMDAS:

- **P**arentheses / **P**arentheses
- **E**xponents / **E**xponents
- **M**ultiplication / **M**ultiplication
- **D**ivision / **D**ivision
- **A**ddition / **A**ddition
- **S**ubtraction / **S**ubtraction

Mnemonic

A common mnemonic to remember the order is "Please Excuse My Dear Aunt Sally."

Important Notes

- **Multiplication and Division have equal precedence.** If an expression has both multiplication and division, they are evaluated from left to right.
- **Addition and Subtraction have equal precedence.** They are also evaluated from left to right.
- **Use parentheses to override the default order.** Parentheses force JavaScript to evaluate the expression within them first.

Example with Parentheses

```
let num = (5 + 5) * 10;
console.log(num); // Output: 100
```

In this case, the parentheses ensure that the addition is performed before the multiplication.

Why is the Order of Operations Important?

- **Consistency:** It ensures that calculations are performed the same way every time, regardless of who writes the code or where it runs.
- **Predictability:** It makes code behavior predictable, which is essential for writing reliable programs.
- **Avoiding Errors:** Understanding the order of operations helps prevent logical errors that can lead to incorrect results.

VIDEO:

In this guide, we are going to go back to some of your earlier math days and we're going to talk about the order of operations. Because the order that javascript is going to interpret your computations is very important and it may or may not be intuitive to you depending on your own math experience.

I want to go through a basic example of this first and then we'll walk through what that order is.

If I create a variable here called num and if you notice I'm back in code pen you can do all of this in codepen or you can do it in the javascript console or on a file on your system. It's completely up to you. But I like the way that this is going to output everything so I'm going to store it in a number and then I'm going to print that number out.

So here I'm going to say five plus five times ten. And if I console log number out what do you think that this will output? Well, if we read this from left to right this would be five plus five which is ten times ten. So it should be a hundred. But is that what we're going to get? hit run.

```
var num = 5+ 5 * 10; // 55
```

You can see we get 55 and that is because of the order of operations that the javascript and for the most part pretty much every programming language, that I've ever worked with follows. That is, that it does not read it from left to right, instead, it dissects it and it looks to see each one of the operators, and then it puts a priority based on that.

The way that it works, is first five is multiplied by 10, which would equal 50. Then the 50 added to 5. That's where we get 55. In the comments, I'm going to say this is called PEMDAS and if you are from the UK or in Europe I have also seen this being called PEDMAS.

We'll talk about what the difference is here in a second. For the how we're going to work with it, you will see that it really does matter. Both work perfectly fine. There is also another sentence I learned when I was a kid and it is. Please Excuse My Dear Aunt Sally and if you notice this is an acronym PEMDAS and it's just a mnemonic to be able to memorize that order. What does this represent?

Well, I'm going to use our knowledge of multi-line comments. And let's actually make this all a multi-line comment.

```
/* Order of Operations
   PEMDAS -> PEDMAS
   Please Excuse My Dear Aunt Sally
*/
```

This is the order of operations. There are both acronyms and a mnemonic.

Now let's go and actually type it out. The P stands for parens. So parentheses are going to be the very first operator that is going to get touched.

First, it's parenthesis, then it's exponents, followed by multiplication, and then division, and then addition, and then subtraction.

```
/*
Parenthesis
Exponents
Multiplication 5 / 5 * 4
Division
Addition
Subtraction
*/
```

This is where we get the, p e m d a s and that is the order.

Let's take a larger example instead of num just equaling this, let's actually just wrap some of this up. We're going to say parentheses will say division let's say a six to the power of two. And then let's do subtraction of one. So this is quite a bit of computation right here.

```
var num = 5 + (5 * 10) / 6**2 - 1;   // 5.38
```

If I run this you can see the answer is 5.38.

The Answer itself doesn't really matter, it's more of I want to show you exactly what the flow is going to be here.

Now that we know this order, then you might guess that the very first time that this hits the first thing that's going to get run is this five times 10. So this is going to be at 50. Now we are going to go to the exponents, we're going to have 6 * 6 which is going to be 36 then we're going there is no more multiplication because multiplication was inside of here. So then we're going to go into divisions, we are going to have 50 divided by 36 and then we'll have 5 on then we'll go down the line.

It eventually ends up just equaling five point three. But this is the order that it follows. I definitely recommend for you to be able to learn how that works. Because if you did run into a scenario where you were performing some type of computation but you didn't organize your operators in the right way then you might end up with something that would be considered a logical error.

We have multiple kinds of errors in programming, we have errors where the system just doesn't run. Where we get an actual error, but we also have logical errors where everything appears to have worked. But the programming logic was not accurate and because of that, the output was not accurate. So it's very important to keep in mind that you are watching not just for errors related to your program running but also ones where it's running but giving the wrong output.

That is something that you see quite a bit especially with new developers, as they're learning is programs that run but may have some odd behavior because of things like this because of the order of operations. So I definitely recommend for you to explore that a bit. Be able to see how you can change it. Test out what happens when you change up the order when you change things from left to right and then see exactly if you understand that.

Now the one thing I will say you may wonder why we have two different ways of reading this we have PEMDAS and we have PEDMAS. I actually learned it with, Please excuse my dear Aunt Sally. Which is the first way and if you grew up in the U.S. then there's a good chance that that is the way you were taught as well.

However, I have spoken with a number of individuals from European countries and they were taught this way. The reason why both of them are accurate and both of them work perfectly fine, is that when you come to multiplication and division the order does not matter. In this case, it will read it from left to right.

In other words, if I have something like 5 / 5 * 4. This is going to be read from left to right. Let's actually test it out just to make sure that I'm not going crazy, or teaching you wrong. Now technically if it does go from left to right this should equal 1. Then 1 times 4 should equal 4. So our output should be four. And if we run this you can see it is four.

```
var num = 5 / 5 * 4;    // 4
```

Even though multiplication in our mnemonic does come before division, that side of it doesn't matter. If you have multiplication afterward it's not going to get run, it's going to be whatever is from left to right.

I've taught multiple programming courses and this is an interesting topic because I have been told by several people that I've been wrong on both sides.

I had one course where I PEMDAS I had another course where I taught PEDMAS and both times I had people commenting saying I was teaching it improperly. Technically they are both accurate. Once you get to multiplication and division the order does not matter. It just reads it from left to right.

The more important thing is to understand how to organize things with parentheses, exponents, and then understand that order in general. I hope that you now have a better idea or understanding, that the order of operations in javascript is very important and also how it works in your own programs.

```
/* Order of Operations
   PEMDAS -> PEDMAS
   Please Excuse My Dear Aunt Sally
   Paranthesis
   Exponents
   Multiplication 5 / 5 * 4
   Division
   Addition
   Subtraction
*/
```

```
var num = 5 + (5 * 10) / 6**2 - 1;
var num = 5 / 5 * 4;
```

Coding Exercise

Following the order of operations, make the below problem equal 76

```
number = 6  10  7;
```

MODULE 2.

CONDITIONALS

2.1 JavaScript Conditional Section Introduction

Hi, and welcome to the section on conditionals in JavaScript. Now, conditionals are one of the most fundamental building blocks of any kind of programming language.

The reason for that is because `conditionals` allow you to have dynamic behavior in your application. This is also something that you can have an analogy in a real-life scenario. We deal with conditionals all day long.
Not even in regards to programming, but if you're driving down the street and you see that you come up to a red light, that is actually you running a conditional in your head where you say: "if the light is red then I'm going to stop."

Conditionals in programming work very similarly. Let's say that you're building out a program for a rental car company, and the rental car company says that if the driver's age is under 25 years old: they're not allowed to rent that car. That is a conditional.

Now, we're going to start off with those basic types of scenarios, and then we're also going to extend it into some more advanced kinds of concepts, such as compound conditionals. To take that rental car, for example, say that you have a policy where you can rent a car if you're 25 years through 80 years old.

Well, you can actually build a compound conditional that checks for both those scenarios. So it can make sure that a data point fits inside of a pre-determined area. At the end of the section we're going to extend our knowledge and we're going to talk about `switch statements`. So we're going to spend the majority of our time working with `if-else conditionals`.

Those are going to be what you're using probably about 95% of the time. There also is this concept in JavaScript called a `switch statement`, and so we're going to examine how you can work with those. I'm also going to use an example from a real-life project that I personally built, so you can see when you'd want to use one type of conditional versus another.

2.2 Basic Syntax for Using Conditionals in JavaScript

This guide walks through the syntax for using conditionals in JavaScript. Including examining the full set of comparison operators.

THEORY:

Conditionals

Conditionals allow you to control the flow of your program based on conditions. The `if` statement is the most common conditional.

```
let age = 12;
let ageTwo = 15;

if (age === ageTwo) {
  console.log('They are equal');
}
```

Comparison Operators

Comparison operators compare two values and return a boolean (`true` or `false`).

- **Equals (==)**: Checks if two values are equal (loose equality).
- **Strict Equals (===)**: Checks if two values are equal and of the same type.
- **Not Equals (!=)**: Checks if two values are not equal (loose inequality).
- **Strict Not Equals (!==)**: Checks if two values are not equal or not of the same type.
- **Greater Than (>)**: Checks if one value is greater than another.
- **Greater Than or Equals (>=)**: Checks if one value is greater than or equal to another.
- **Less Than (<)**: Checks if one value is less than another.
- **Less Than or Equals (<=)**: Checks if one value is less than or equal to another.

Best Practices

- Use === (strict equality) whenever possible to avoid unexpected type coercion issues.
- Use clear and concise conditions to make your code readable.

Example: Age Check

```
let age = 25;

if (age >= 25) {
  console.log('Old enough to rent a car');
}

if (age < 10) {
  console.log('You can eat from the kid menu');
}
```

VIDEO:

As I mentioned in the introduction this section is going to be all about conditionals.

Conditionals give us the ability to look at either a couple of values or even multiple ones. Three, four, or five depending on what you need to check against and see how they are related to each other. We can see if they're equal to each other if one is greater than the other.

We can check to see if they are explicitly not equal to each other and then we also in javascript have the ability to check to see if they are of the same type as well. Let's talk about how we can test this and what the syntax is.

Here I'm going to do a pretty basic one and say var age and we'll say 12 years old and then we'll say var ageTwo this one will say is equal to 15.

```
var age = 12;
var ageTwo = 15;
```

Now what the syntax is for checking for seeing if items are equal, we can say if age is equal to age too and then we use curly braces say console.log they are equal.

So now if we run this, this is should run absolutely nothing because they're not equal. Now if I change this to 12 hit-run you can see it prints out that they are equal because the values were changed.

This is the basic syntax. Now in addition to equals, we also have what's called triple equals(===). If I do three equals this is going to check to see (it's called strict equal) is what the name of the operator is, or the comparison operator.

Now if I hit run you can see it prints out once again they are equal. Everything there pretty much makes sense.

Now if I wrap ageTwo up as a string and hit run again nothing prints out. The reason for this is because strict equal checks not just on value but also type. Now if for some reason you didn't care about the type you just cared to see if they were equal in value and you wanted to allow for javascript to do its own typecasting. You can use two equals.

If I hit run you can see that it prints out. They are equal so that's what the difference is between two equals and three equals. Just to give you a little bit of an idea, this is considered a poor practice. It's very rare when you will ever want to use two equals. The reason for that is because you may run into an issue where you do two equals and that's going to allow everything inside of these curly brackets to run and some of those functions that you have inside the curly brackets. They may be

expecting ageTwo, to be a number. If you allow this to go in and you allow this to be true then essentially what is going to happen is you're going to end up with a bug because you're going to have ageTwo that is acting kind of like an integer. But at the end of the day, it's still a string, The best practice is usually to have three equals. That is the most standard way of doing it.

```
if (age === ageTwo) {
  console.log('They are equal');
}
```

Let's talk about some of the other ones. We have our triple equals and now let's do something similar. Now we're going to do some of the different operators. And the reason why I'm commenting it out is that you can. You'll be able to have these all in the show notes. So never get checked to see if age is equal or is not equal to ageTwo. Here the syntax for doing that is going to be a bang followed by equals followed by whatever you're trying to compare it to.

```
if (age !== ageTwo) {
  console.log('Not equal');
}
```

consol.log not equal now if we run this should print out not equal. Everything there is working nicely, now just like we have our three equals four having a regular equal versus a strict. There also is another option here that is going to be strict not equals. If I run this you can see right here it's going to print out not equal but it was not equal before.

This has to deal with the same exact type of scenario. This returns true if the operands are the same type but they're not equal or they're of a different type. In other words, if I say this or 12 here and so if I run this again it still prints out that they're not equal because at the end of the day they're of a different type. So everything there is working properly and once again it's considered kind of the best practice to do the bang which is an exclamation mark followed by two equals.

We've covered four of them so far we've covered equals, strict equals not equals, and then strict not equals.

Now we have just a few more to go through. Let's set up an application that checks to see if someone is old enough to rent a car. I'm going to say if age and we'll say is greater than or equal to (which this is the right way of doing it) 25.

This is a way of saying greater than or equal to versus just greater than. Let's say greater than or equal to, old enough to rent a car.

```
if (age >= 25) {
  console.log('Old enough to rent a car');
}
```

Now if I run this nothing prints out and it shouldn't. If I change age to be 25 and run it again it says old enough to rent a car. Notice that 25 is not greater than 25 it's equal to and that's where we have greater than or equal to. Now if I just did greater then and we hit run nothing prints out, that is because the difference is between greater than, and greater than or equal to, is greater than or equal to is actually looking for two conditions. It's checking to see, is this value greater than this other value or is it equal to? That's something that's, almost kind of like we said something like

```
if (age == 25) {
console.log('Old enough to rent a car');
}
```

We ran that conditional it would have printed out the same thing but because we have the ability to do greater than or equal to then it simply works. Now, this does not have a strict version. We only have the ability to do greater than or equal to.

Then we also have the opposite of that. We'll say if age is less than 10 and then in the console log "you can eat from the kid menu".

```
if (age < 10) {
console.log('You can eat from the kid menu');
}
```

Now if we run this nothing happens. But if we go back and change the age to 8 then it says you can eat from the kid menu. Also if you say less than or equal to and we change the age to 10, if I hit run again it now says you can eat from the kid menu because it was not less then but it was equal to.

That's the full list of the comparison operators we have. Starting at the top we have equals and strict equals. We have the other one which is going to be not equal to. And then we have strict equal to. Then we have greater than or equal to. And then we have less than or equal to. Those are going to be what you're going to be placing inside of your conditionals and the left side and the right side these are both called `operands`.

These are the values whether they're variables or just straight hardcoded values that the different comparison operators are going to be comparing.

One final note, notice that it's not required that you use a variable. There are going to be times where you are trying to compare two different dynamic values. In that case, you will be using variables. But there's also going to be times where you do want to hard code something in like we did when we added a number and you're able to do both of those things when using comparison operators in javascript.

```
var age = 10;
var ageTwo = '12';

if (age === ageTwo) {
  console.log('They are equal');
}

if (age !== ageTwo) {
  console.log('Not equal');
}

if (age >= 25) {
  console.log('Old enough to rent a car');
}

if (age <= 10) {
  console.log('You can eat from the kid menu');
}
```

Coding Exercise

Create a conditional that returns true, using the starting code below.

```
answer = false;

if (input your conditions here) {
 answer = true;
}


NOTES:
```

```
/*
# EXPRESSIONS (basics, grouped) and OPERATORS:
Expressions: Piece of code which does "something".
Operators: The symbols needed to do that "something".

### Types:

- PRODUCTIVES
Those which do something to cons vars functs.
E.g. the EQUAL operator from a let sentences

- EVALUATIVES
E.g., an arithmetic calculations.

## OPERATORS:
Symbols which produces an action.

BY OPERANDS NEEDED:
- Unary opers: typeof, delete, return,
Unary just because the needed of ONE operand.

- Binary opers:
Need TWO operands.

- Ternary opers:
Need THREE operands.

BY STRUCTURE:
- BASIC: this, typeof, return, ...

- GROUPED/COMPLEX: if elif return expressions on a function, ...

BY TYPE OF DATA:
- Arithmetic opers: +, -, /, *, %, **, ...

- Assignment opers: =, +=, =+, -=, =-, ...

- Comparison opers: for BOOLEANS: ==, ===, !=, !==, <, >, >=, <=, ...

- Bitwise opers: &
AND oper is tricky, it converts the given value to numeric.
Works in both ways, for Integers, and Big Integers (those more than
32bit lenght).
The result is a lack of accurace on calculations.

- Logic opers: &&, ||, !
Used for combining operators to give a boolean value.

- String opers: +,
Concatenating.

- Relational opers: in, instanceof, ...
Evaluates the relationship between objects.

BY ORDER:
As explained on PEMDAS > PEDMAS


## CONDITIONAL Operators: Ternary operators:

A conditional evaluates and decides which code run, or not, (if, else,
elif, ...)
depending on which Conditional Operators ( ==, ===, !=, !==, <, >, <=,
>=, we need to set.

== - LOOSE EQUALITY
```

```
=== - STRICT EQUALITY
!= - LOOSE INEQUALITY
!== - STRICT INEQUALITY
> - GREATER THAN
< - LESS THAN
>= / <= - Greater than or Equal / Less than or Equal



=== and !==, when parsing values, can help preventing weird behaviours
on exec if the operands (given data) isn't correct


*/

/*
Adding notes to truth tables and arith. opers:

=== Strict equal. DOES NOT COMPLAINS with booleans

*/
```

2.3 If/Else Conditionals

This guide shows how to implement if/else conditionals in JavaScript in order to give programs dynamic behavior.

THEORY:

`if/else` Conditionals

`if/else` statements allow your program to execute different blocks of code based on a condition.

- `if` **block:** Executes if the condition is true.
- `else` **block:** Executes if the condition is false.

Example: Age-Based Menu

```
let age = 30;

if (age <= 10) {
  console.log('You can eat from the kid menu');
} else {
  console.log('Adult menu time for you');
}
```

In this example, if `age` is less than or equal to 10, the first message is printed. Otherwise, the second message is printed.

How it Works

1. **Evaluation:** JavaScript evaluates the condition inside the `if` statement's parentheses.
2. **Execution:**
    - If the condition is true, the code inside the `if` block executes.
    - If the condition is false, the code inside the `else` block executes.

Benefits of `if/else`

- **Dynamic Behavior:** Allows your program to respond differently based on conditions.
- **Improved Logic:** Helps create more complex and nuanced program logic.
- **User Experience:** Enables you to tailor the user experience based on user input or application state.

Example: Login Check

```
let isLoggedIn = true;

if (isLoggedIn) {
  console.log('Welcome back!');
} else {
  console.log('Please log in.');
}
```

Important Notes:

- `if/else` statements provide a way to execute different code based on a condition.
- The `if` block executes if the condition is true, and the `else` block executes if the condition is false.
- `if/else` statements are essential for creating dynamic and responsive JavaScript programs.

VIDEO:

Now that you have a basic idea on the syntax for using `conditionals` and also a good idea on the full list of the `comparison operators`.

Let's talk about how we can give our programs more of a dynamic type of behavior. So far we've just talked about implementing `if statements` an if statement isn't too handy without also having the ability to give another option. The ability to have an if or an else, I love the way that they're described because you can read it almost like plain language. Where you can say if such and such is true. If this condition is met I want you to run everything inside of this section. Else, if not then I want you to show everything or run everything in this other section.

Let's go through an example. I'm going to say var age is equal to 30. Now if I say age is less than or equal to 10 then we'll console log. "You can eat from the kids' menu".

```
var age = 30;

if (age <= 10) {
  console.log('You can eat from the kid menu');
}
```

Right now if we run this, nothing is going to happen because this is not going to look in this condition and find a true statement. Remember the way that conditionals work is they check to see if a certain kind of condition is true or not. In this case, the condition is age being less than or equal to 10.

What javascript does, is it comes in here and it looks inside of these parentheses and it says, that is not true so I'm going to ignore everything inside of here. Everything from the curly bracket down I'm going to ignore and that's fine, in certain situations. There are many times where I use just an if statement just by itself.

However, in this situation and in many other situations you want to have some other kind of condition you want to have, this is what I want you to do if the conditional is true but I may want you to do something else if it's false. Here what we can say is just `Else` right after the curly brackets and then here we'll console log just copy up and instead of saying "you can eat from the kids' menu" say "adult menu time for you".

```
if (age <= 10) {
  console.log('You can eat from the kid menu');
} else {
  console.log('Adult menu time for you');
}
```

Now if I run this is going to go through and it's going to skip over the first condition right here, it's going to skip over everything inside the curly brackets, because this is not true. So because that's not true. It skips down to the else and then it's going to run everything inside of the last brackets. Now if I change the age to be 8, then hit run again.

Now you can see it says "You can eat from the kids' menu".

So that is how you can implement a basic `IF ELSE` condition inside of javascript.

In the next guide, we're going to go through how we can add more complexity here and how we can set up a full set of scenarios for our conditions.

```
var age = 30;

if (age <= 10) {
  console.log('You can eat from the kid menu');
} else {
  console.log('Adult menu time for you');
}
```

Coding Exercise

Write a condition that returns `true` if you have more than 50 watermelons.

```
function watermelonParty() {

 watermelons = EnterYourNumberHere;

 if (WriteYourConditionsHere) {
 return true;
 }
}

watermelonParty();
```

NOTES:

```
/*
adding notes:
Ternaries don't use if/else conditionals, they have it implicitely
inside
*/
```

2.4 Compound Conditionals

This guide examines how to implement compound conditionals in a JavaScript program.

THEORY:

`if/else` Statements

Compound conditionals allow you to chain multiple `if` statements together with `else if` clauses to test a series of conditions.

- `if` **block:** Executes if the first condition is true.
- `else if` **block(s):** Execute if the preceding condition is false and the current condition is true.
- `else` **block:** Executes if none of the preceding conditions are true.

Example: Age-Based Permissions

```
let age = 30;

if (age <= 10) {
  console.log("You can eat from the kid's menu");
  console.log("You are not old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 16 && age < 25) {
  console.log("You can not eat from the kid's menu");
  console.log("You are old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 25) {
  console.log("You can not eat from the kid's menu");
  console.log("You are old enough to drive");
  console.log("You are old enough to rent a car");
}
```

Explanation

- The code checks the first `if` condition (`age <= 10`). If true, it executes the corresponding block and skips the rest.
- If the first condition is false, it moves to the `else if` condition (`age >= 16 && age < 25`). If true, it executes that block.
- If none of the `if` or `else if` conditions are true, it executes the final `else` block.

Logical Operators

Compound conditionals often use logical operators like `&&` (AND) and `||` (OR) to combine conditions.

- **`&&` (AND):** Both conditions must be true for the combined condition to be true.
- **`||` (OR):** At least one of the conditions must be true for the combined condition to be true.

Best Practices

- **Be Explicit:** Avoid relying solely on the `else` block for complex logic. Use explicit `else if` statements to handle specific scenarios. This prevents unexpected behavior due to invalid or missing data.
- **Order Matters:** The order of your conditions matters. Place more specific conditions before more general ones.
- **Keep it Readable:** Use indentation and comments to make your compound conditionals clear and easy to understand.

Abstract

- Compound conditionals allow you to handle multiple scenarios with `if/else if/else` statements.
- Logical operators like `&&` and `||` combine conditions.
- Being explicit with your conditions improves code reliability and maintainability.

By mastering compound conditionals, you can create more sophisticated and dynamic JavaScript programs that respond effectively to various situations.

VIDEO:

In the last guide, we walk through how to have if-else statements with our conditionals. This is a very common pattern and you're going to be using if-else blocks just by themselves quite a bit.

However, javascript also has the ability to set up multiple scenarios. This is called the if-else if type of conditional. What I want to do is to set up a full range of scenarios. So this is going to check to see if, you are old enough to eat from the kids' menu, if you're old enough to drive, and if you're old enough to rent a car. Then set up different types of age ranges. This is where our conditional is actually going to search within a range and then output whether the user or whoever has that age is able to perform those tasks.

Right here we're going to keep this part the same. So if age is less than 10 then you can eat from the kids' menu.

```
if (age <= 10) {
  console.log("You can eat from the kid's menu");
```

But I'm also going to add a few other ones. This is going to say you are not old enough to drive. And then also you are not old enough to rent a car. So instead of an if-else statement, we're going to get rid of that entirely. I'm going to say else if and put in a range, the way that you can set up a range in javascript. Is I can say age is greater than 10. Then we use AND by putting in double ampersands. Age is less than 25. Then we want this to happen.

```
if (age <= 10) {
  console.log("You can eat from the kid's menu");
  console.log("You are not old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 16 && age < 25) {

}
```

So let's review what is going on here.

The syntax for having another conditional. So not just an else. The else is kind of like something that is always going to run if this is false. Now what we're doing is we're saying in a scenario where the age is less than or is greater than 10 and the age is less than 25. What our program does, is it's going to check the left conditional and then if this is true. It's going to come and check the right conditional. Now if both of them are true everything that happens inside of the curly brackets is going to go on now if it comes across and sees that age is greater than 10 and so that would be true but age is not less than 25. Say, this is for someone who's 40 years old then this is going to be false and the entire thing is going to be false and it's going to go on to the next conditional if there is one.

That's how a range works. So I'm just going to copy all of the console logs. "You can eat from the kids' menu" is going to change to, "You can not eat from the kids' menu".

The next one, "You are not old enough to drive" is going to be "You are old enough to drive". And then the last one is "You are not old enough to rent a car". This one is still the case.

We've already done a decent amount of work. Let's see if this is going to work.

```
var age = 8;

if (age <= 10) {
  console.log("You can eat from the kid's menu");
  console.log("You are not old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 10 && age < 25) {
  console.log("You can not eat from the kid's menu");
  console.log("You are old enough to drive");
  console.log("You are not old enough to rent a car");
```

If I run this, it says "you can eat from the kids' menu". "You're not old enough to drive". "Not old enough to rent a car". That's because the age is 8 and it hits age is less than or equal to 10. So this all worked.

Now let's change up this age to be somewhere else.

Now we're going to say age = 16 and it has to be greater than or equal to 16 `else if (age >= 16 && age < 25)`. It won't work at 10. I don't think we want a lot of 10-year-olds on the road.

Now if we have a greater than or equal to 16 if we run this, then it says "you cannot eat from the kids' menu". That's good. "You are old enough to drive" and then "you are not old enough to rent a car". Everything there is working perfectly. Now let's change it up one more time before we continue on and we'll say someone who's 30 years old if I run. Nothing happens, that's because the age of 30 is greater than 16 but it is also greater than 25. So it doesn't match both of these conditionals so in order to get this to work there are a couple things we could do.

One we could put just a regular else so we could put an else here and everything would work.

```
else {
console.log('something');
}
```

And we'll see if this works when it is 30.

Now if this runs you can see it prints out "something" and that is fine, in some types of conditions. However, if you want to be more explicit. You want to make sure that what you're looking for is also what you're going to find. Then we would be better to do another if-else.

The reason why I would want to do that is let's imagine a scenario where age is an even number but it's something else. So say it you know it's a string or some kind of bad piece of data. This console log may actually come and happen or anything inside of here may happen. It would be better if we were explicit, this guarantees that we're not just going to fall into a default. But instead, we're going to check to look for another specific type of condition.

Now what I can do is say we don't need a second one. We can just say if age is greater than or equal to 25. I want you to do all of this. Here we can say "you cannot eat from the kids' menu", "You are old enough to drive", and "you are old enough to rent a car". If we run this again you can see that it worked. We can also be more confident in how that is being processed. If somebody puts a bad piece of data as our age, it is not going to hit our default.

That is a little bug, that I have seen in a number of programs where the developer didn't think it all the way through and they just had three conditions and they assumed that if 1 and 2 were not met then that last one is fine to just throw it in the else block because everything could fit inside there.

However, there are plenty of times where I've seen that lead to some very disturbing side effects. Imagine a scenario right here where you were building a program said this. This essentially, a validation little code library. Imagine you put this in a web program and you put this last one just as regular else.

And somebody who just happened to not have a name so era an age they went in there and so say it was set to null or something like that and instead of it printing out an error message, instead it told them they could rent a car that would be a really bad thing so make sure whenever you're building these types of conditionals that you're examining all of the possible scenarios and how you want your program to behave. In review, this is the syntax, and essentially the best way to write a multiple type of condition including being explicit even when you think that everything is going to fall inside of a specific condition.

```
// We recognize that there is a hole is this code logic covering ages
11 -15. As was discussed in the lesson this does happen.
```

```
// Try playing with the code and fixing the hole in the logic for
practice.
var age = 30;

if (age <= 10) {
  console.log("You can eat from the kid's menu");
  console.log("You are not old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 16 && age < 25) {
  console.log("You can not eat from the kid's menu");
  console.log("You are old enough to drive");
  console.log("You are not old enough to rent a car");
} else if (age >= 25) {
  console.log("You can not eat from the kid's menu");
  console.log("You are old enough to drive");
  console.log("You are old enough to rent a car");
}
```

Coding Exercise

Create a condition that allows a 15 year old to get a permit, but can't get a license.

```
function kid() {
    age = GiveYourKidAnAge;

    if (EnterYourConditionsHere) {
        return true;
    }
}
```

2.5 Building Switch Statements to Check for Data Types

In this guide you'll learn how to work with JavaScript switch statements. The example we'll walk through will examine how to check for the data type of a variable.

THEORY:

Switch Statements

Switch statements provide an alternative to `if/else if/else` chains for handling multiple conditional scenarios. They are particularly useful when you need to compare a single value against several different options.

Syntax

```
switch (expression) {
  case value1:
    // Code to execute if expression === value1
    break;
  case value2:
    // Code to execute if expression === value2
    break;
  // ... more cases
  default:
    // Code to execute if none of the cases match
}
```

Example: Data Type Check

```
let dataPoint = 'Hi there';

switch (typeof dataPoint) {
  case "string":
    console.log("It's a string");
    break;
  case "number":
    console.log("It's a number");
    break;
  case "boolean":
    console.log("It's a boolean");
    break;
  default:
    console.log('No matches');
}
```

Explanation

1. **Expression:** The `typeof dataPoint` expression is evaluated.
2. **Case Matching:** The result of the expression is compared to each `case` value.

3. **Execution:** If a match is found, the code within that `case` block is executed.
4. **Break:** The `break` statement prevents the code from "falling through" to the next case.
5. **Default:** If no match is found, the code in the `default` block is executed.

Benefits of Switch Statements

- **Readability:** Can be more concise and readable than long `if/else if/else` chains, especially for multiple comparisons.
- **Organization:** Provides a clear structure for handling different cases.
- **Efficiency:** Can be slightly more efficient than `if/else if/else` in some scenarios.

Important Notes

- **Strict Comparison:** Switch statements use strict equality (`===`) for comparisons.
- **Break Statements:** `break` statements are crucial to prevent fall-through behavior.
- **Default Case:** The `default` case is optional but recommended for handling unexpected values.

Abstract

- Switch statements provide an alternative way to handle multiple conditional scenarios.
- They are especially useful for comparing a value against a set of options.
- Use `break` statements to control code flow and prevent fall-through.
- Consider using a `default` case to handle unmatched values.

VIDEO:

Now that we've covered multiple ways that we can implement conditionals in javascript. I want to give you an alternative approach we focused on building if, and if-else, and then if else if type of conditionals. But now there is another one that is good to have an understanding of. It's not quite as popular as if-else you're going to see that probably the most in the wild but this still is something that's good to know and there are scenarios where it works quite well.

It's called a case statement. A case statement gives you the ability to build your own scenarios. Now, this doesn't work quite as well as if-else in say for the program that we built with that age scenario type tool usually you'll see Case statements when there is just a single set of scenario options and it's checking just to see if something is equal or not and if that seems a little bit obscure, don't worry I think it'll make a little bit more sense when we're actually building it out.

So first I'm going to create a variable called datapoint and we'll set the sequel to 'Hi there' just a regular string.

```
var dataPoint = 'Hi there';
```

What our program is going to do, is it's going to take in the data point and no matter what's inside it's going to output what data type that specific piece of value was what that variable was. So the syntax for this is to say switch and then put inside whatever type of value you want. In my case, I want to do type of data point. Remember what type of does? It checks to see what type of value your data is. So if I open up the console and I say `typeof 'Hi there';` this is going to say it is a string.

That's all we're doing. We want to see whatever the data type is, that we're going to be using. So that is going to return String number it's going to return whatever the data type is. Then we're going to follow that with curly brackets and inside of here, we're going to call our scenarios. We're going to create them.

The first one is a case and then string followed by a colon and then anything inside of this is what we want to happen if the typeof returns, in this case, a string. So here just go to a console log and say it is a string. And we're excited about it. So we put an exclamation mark. So the next thing you have to do is call a break. What this is going to do is it's going to say that we found our case. We have our scenario, now break and skip all of the other scenarios. So if you don't put break in there you're going to run into some weird bugs.

```
switch (typeof dataPoint) {
  case "string":
    console.log("It's a string");
    break;
```

Next, we want to set up another case. Here we're going to say case for number and give a colon and now everything inside of it is going to be pretty much the same except just the output. So I'm going to put this inside of it. But instead of saying it's a string we're going to say it's a number. Now copy all of that for the next one.

Make sure you have your indentation proper It should all be lined up just like I have it. And for this one I say boolean and this will say it's either true or false because those are the only options that it can have.

```
 Case "boolean":
    console.log("It's a boolean");
    break;
```

And that is going to be it for this moment. I'm going to show you how to do something else later.

This is everything that we'll need to get this working. Right here we have data point 'Hi there'. So this should match the string. If we did everything right. So if I hit run it matches so it says it's a string. So everything there is working properly. If I change our datapoint and change it to say 5 and hit run. Now it's going to say that it's a number.

Now there are other times where we might want to have a default. The usual case for that is not to use it as a catchall. Remember when we went through our nested and our compound if-else. Remember I said you don't want to use an else to just catch everything that wasn't caught above. case statements are the exact same thing. Say that you had five different data types you're checking for. You don't want to put the fifth one as the default. You want to declare it the same way we did with our case, but now if you want something that says, say an error message you can create what's

called a default. The default doesn't know what it's going to get, because of that it's going to not require a scenario.

Then we're just going to console log no matches.

```
default:
    console.log('No matches');
```

This one also does not require a break, remember a break is telling the program to end it and the switch statement with the default, is only going to come here if none of the other cases were met. So it doesn't need that default. Let's run it, everything still works exactly the same. If I come up and instead of 5 pass in an object hit run it's going to say no matches. Because an object is not a string a number or a boolean. So that is how you can leverage switch statements in Javascript in order to build your own conditionals.

```
var dataPoint = {};

switch (typeof dataPoint) {
  case "string":
    console.log("It's a string");
    break;
  case "number":
    console.log("It's a number");
    break;
  case "boolean":
    console.log("It's a boolean");
    break;
  default:
    console.log('No matches');
}
```

Coding Exercise

Write a switch statement that always returns: *"number"*

```
function switchStatement() {

    //Write your switch statement within this function

}

switchStatement();
```

2.6 Overview of JavaScript Ternary Operator

Let's walk through the code for the ternary operator, and we're first going to start off by looking at why the ternary operator in JavaScript is necessary.

THEORY:

Why Ternary Operators?

Ternary operators are useful for writing conditional logic in a single line, which is often required in situations like JSX (JavaScript XML) used in React and Vue.js applications.

Example: Conditional Styling

```
<div className={hasPermission ? 'active' : 'disabled'}>
</div>
```

This code conditionally applies the `active` or `disabled` class based on the `hasPermission` variable.

Basic Ternary Syntax

```
condition ? expression1 : expression2;
```

- **`condition`:** The condition to be evaluated (e.g., `age > 25`).
- `expression1`: The expression executed if the condition is true.
- `expression2`: The expression executed if the condition is false.

Example: Age Verification

```
function ageVerification(age) {
  return age > 25 ? "can rent a car" : "can't rent a car";
}
```

Compound Ternary Operators (Nested Conditionals)

While possible, nesting ternary operators for compound conditionals can make code less readable. Use with caution.

Example: Admin Controls

```
function adminControls(user) {
  return user
    ? user.admin ? "showing admin controls" : "You need to be an
admin"
    : "you need to be logged in";
}
```

Explanation

- Checks if `user` exists (`user ?`).
  - If true, checks if `user.admin` is true (`user.admin ?`).
    - If true, returns "showing admin controls."
    - If false, returns "You need to be an admin."
  - If false, returns "you need to be logged in."

Best Practices

- **Keep it Simple:** Use ternary operators for concise conditional logic.
- **Avoid Over-Nesting:** Excessive nesting can harm readability.
- **Consider Alternatives:** For complex logic, `if/else` statements might be clearer.

Important:

- Ternary operators provide a concise way to write conditional expressions.
- They are valuable in situations where single-line conditionals are required (e.g., JSX).
- Use compound ternary operators sparingly, as they can reduce readability.

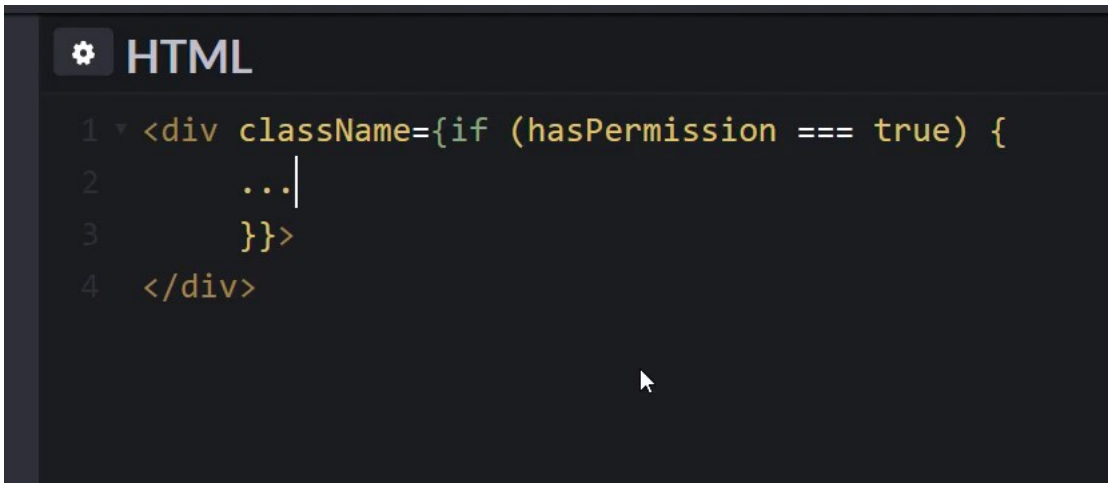
VIDEO:

The reason why I'm going with this approach is because I've seen through the years that the ternary operator can be very confusing, it has a very different syntax than the regular JavaScript conditional or even the Switch statement. And so I want to first show why ternary operators are important to understand because the very first question that I usually get from a student after I have shown them how to use a ternary operator is, "Why in the world would I want to use this really weird looking syntax when I could use a regular if/else conditional or a case statement?" And it's a fair question until you see the rationale for why.

So I'm going to open up the HTML tab right here. And I'm going to write what looks like HTML but it's actually a templating engine called JSX and so if you write any programs or you ever want to write any programs in React or Vue, then you might use JSX in order to write out your templates. So it looks a lot like HTML and for the sake of example, you can just imagine that it's HTML if you've never used it before, 'cause it looks like it.

So I'm going to create a div here. And let's imagine that I'm building out a React project and I want to show or I want to hide a div and let's say, it's a tab and the way that I can do that is by saying, className= and then in curly brackets here, I can write some JavaScript code.

Now, if I want to show or hide this specific div or this tab, or whatever it is, then I need to have some conditional logic. So say, we want to check to see if someone has the right permissions, then I can't do something like this. I can't write a regular conditional and say if, has permission === true and then on another line like this have all of my logic. That is not something that is allowed.



So I'm going to get rid of all of that and let's see what we can do, so this is where the ternary operator comes in. I have to write this all on one line, and what a ternary operator allows you to do is to do that, is to write an entire conditional on a single line.

So here what I could say is, has permission and then I'm going to do a question mark, and then we'll say active and I'm making all of this up right here, this is just an example to show what you may build. Then we're going to get into real examples later on. So I could say active colon and then disabled.

```
<div className={hasPermission ? 'active' : 'disabled'}>
</div>
```

So what I have done here is I've provided a conditional. So this is the same thing as saying, if has permission then I want you to return active and if not, I want you to return disabled. This is the only way or the proper way I should say, for building a conditional in tools like React or Vue so that you can have some dynamic behavior built directly into your HTML and your JSX.

So this is the main reason why ternary operators are so important to learn because if you are building out any kinds of real-world front end applications, you're most likely going to have to build in something like this at some point or another.

Now that you've seen the example, I'm going to comment this out, and now let's go into the JavaScript code, and let's actually go through a real working example. I'm going to start with a basic

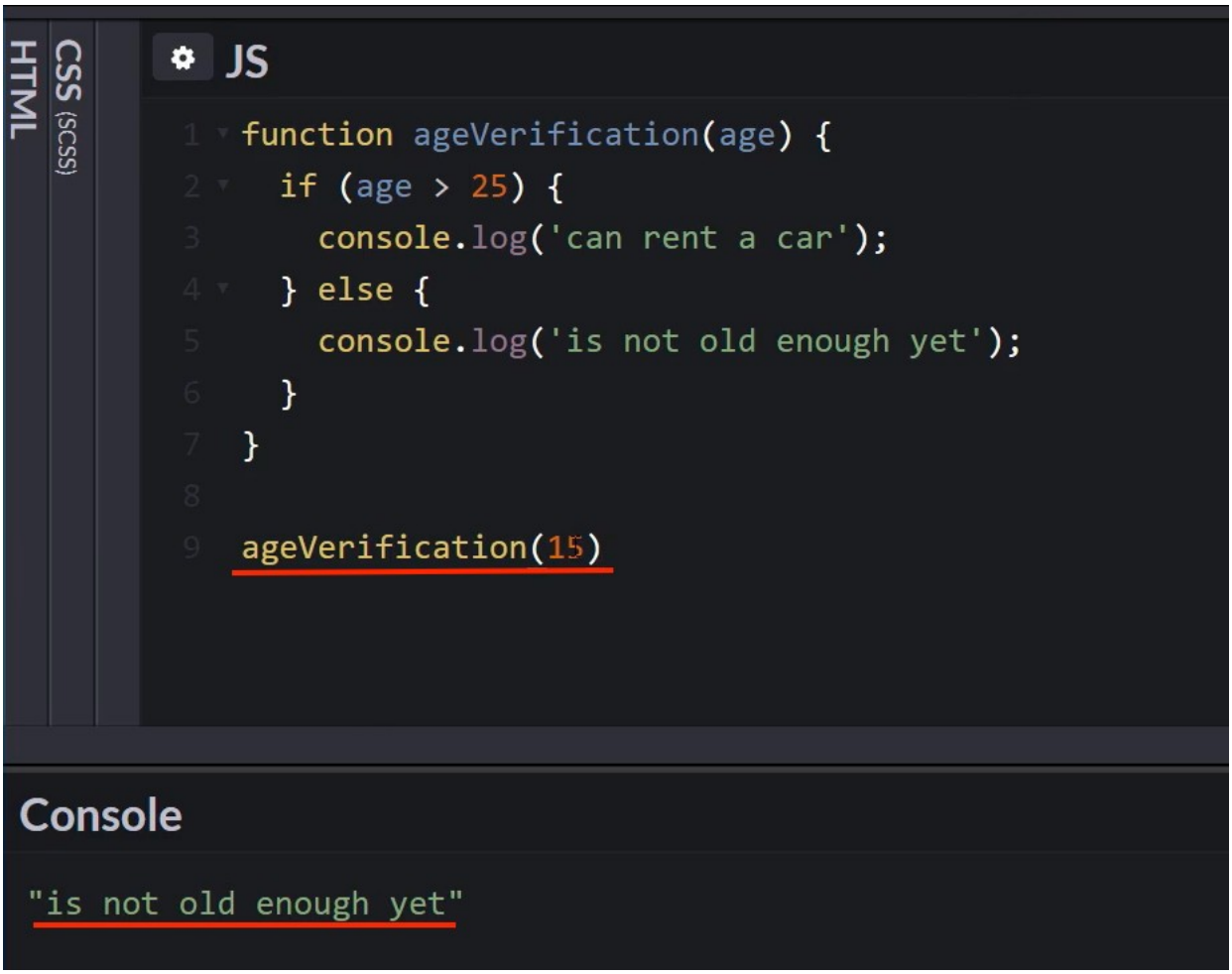
one, and then we're going to go into a more advanced one. So I'm going to start off by creating a function here.

I'm kind of call it age, verification, it's going to take in an age, and then inside of here, I'm going to place the conditional. If you're brand new to JavaScript, and you've never heard of functions at all, do not worry, this is going to be very basic. I'm simply wrapping all the behavior up in a function, so it's easier to call it and print it out down here. So this isn't going to dive into functions and details just a wrapper for what we're going to be doing.

Here I could say, if the age is greater than 25, then I want to console log, can rent a car, and then right here we want to provide an else statement, and then say I want that to be console log is not old enough yet. And that's all that we want this function to do.

```
function ageVerification(age) {
    if (age > 25) {
        console.log('can rent a car');
    } else {
        console.log('is not old enough yet');
    }
}
```

So it's pretty basic then I'm going to call this function by saying age, verification, if I say 15 here, if I save and then run it, it's going to print out that the user is not old enough.



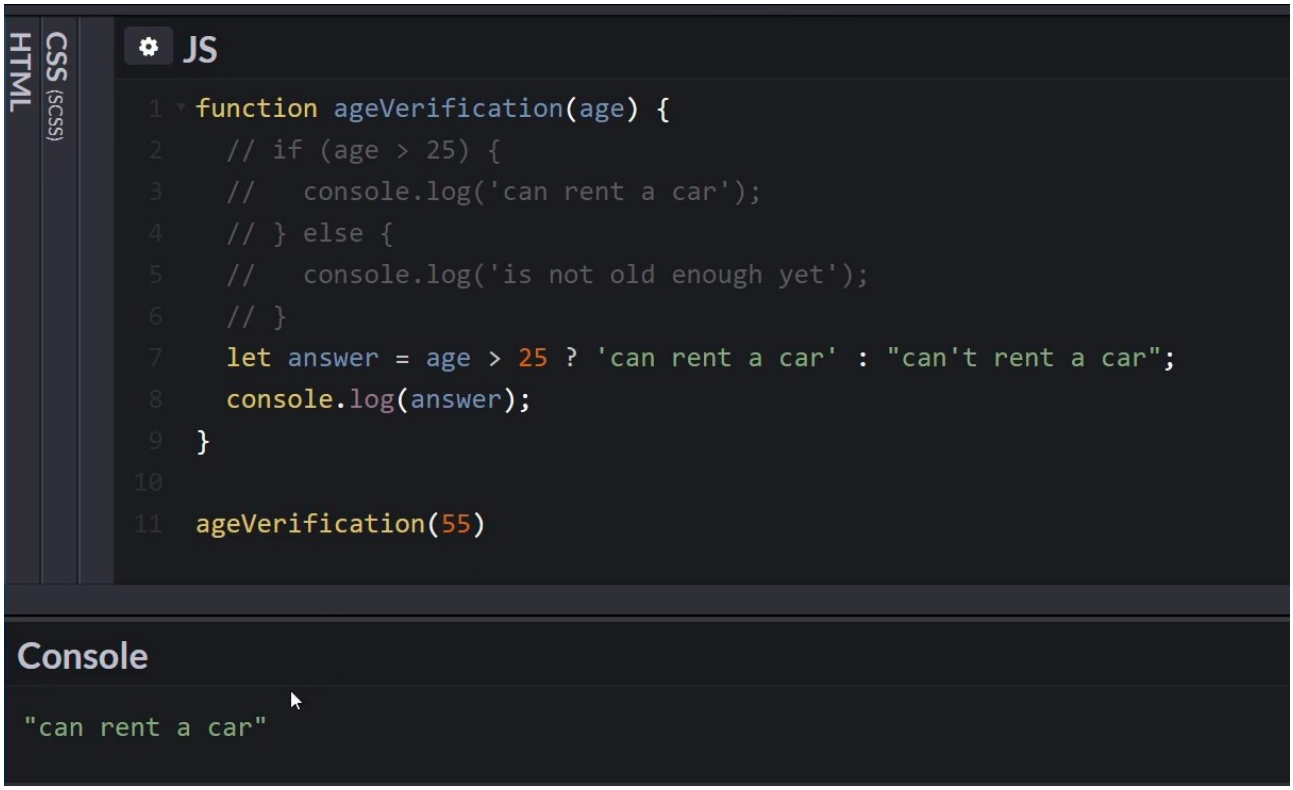
If I change this to 55, and then run it again, now it's going to say they can rent the car.

So this is all working properly. It's an incredibly basic function and this conditional is pretty much as basic as you can get. Now, I did that on purpose. Anytime that I'm wanting to learn something new, especially something that might be a little bit more on the confusing side, I like to start off with a base case. So we're going to simply comment this out and then below here, I'm going to show you the syntax that will allow you to have a ternary operator.

We're going to use the exact same logic, we're just going to switch it up and use it with the ternary syntax. Now the way that you can do this is I'm going to store it in a variable. So I'm going to say, `let answer =` then here, I'm going to say `age > 25`, and then a question mark. Then I'm going to have it say the same thing. So I'll say I can rent a car and then colon can't rent a car. You could put is not old enough, whatever you want on that side.

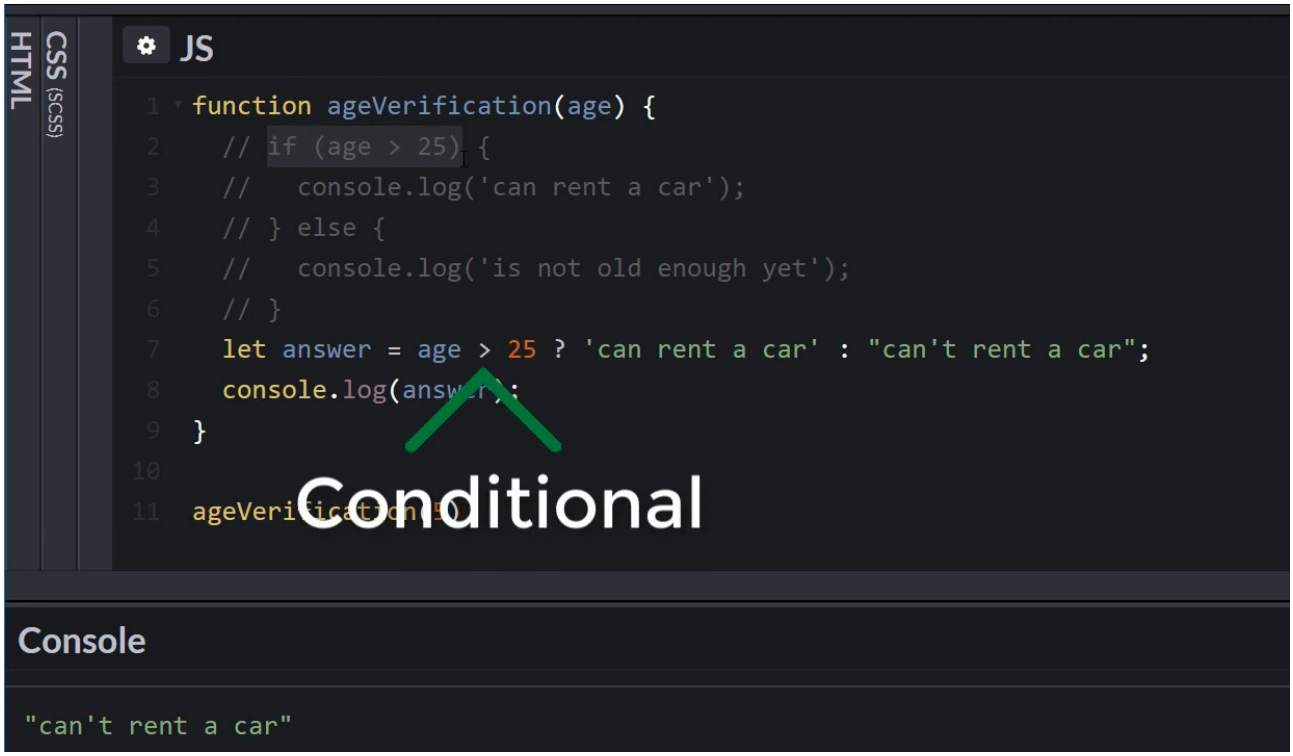
So that's going to store it in a variable and then let's simply print out the value of whatever that variable is. So we'll say console log, answer, and don't worry, I know I typed all of that out pretty quickly we're going to walk through exactly what the mapping is doing and everything.

Let me hit clear, save, and then run it. And now you're going to see it says, "They can rent the car."



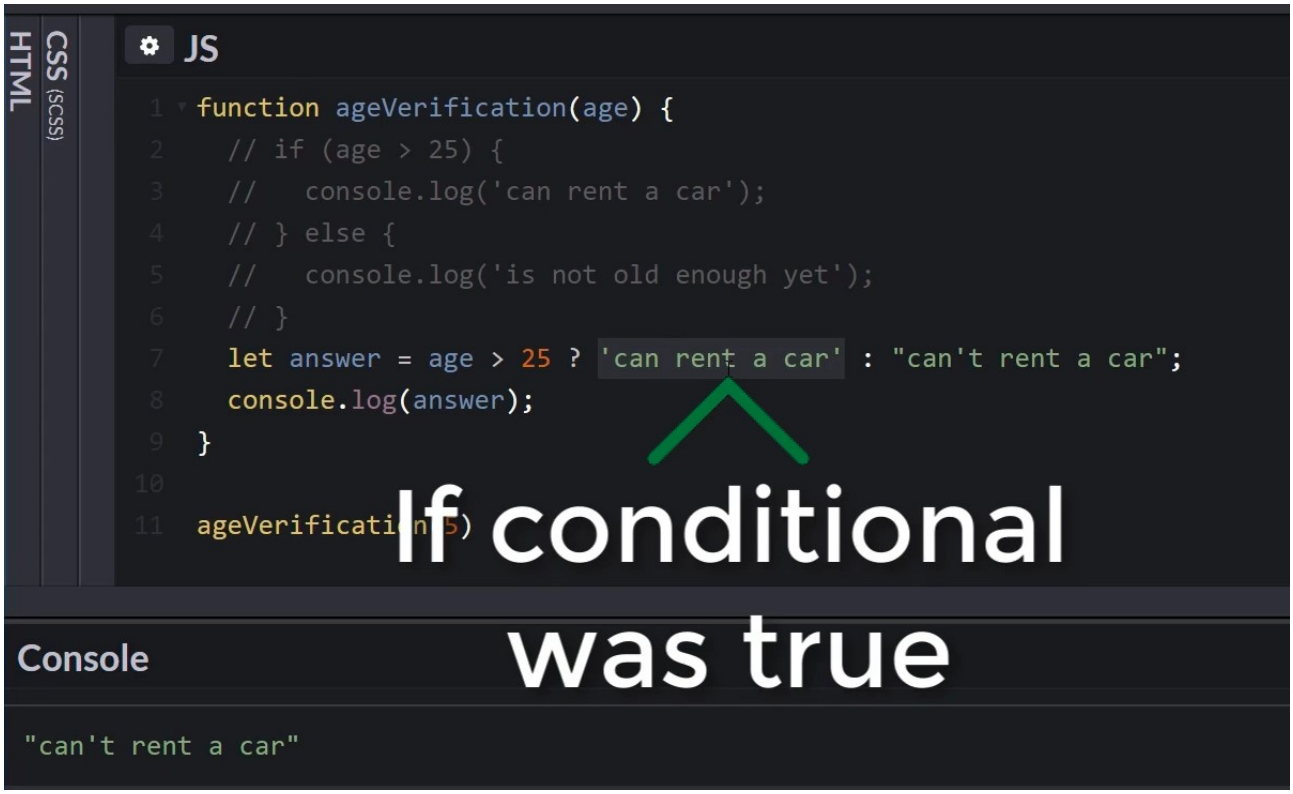
So we're getting the exact same answers as before, if I change this to five years old and hit run, it's going to say they can't rent the car. So this is working perfectly, this is the exact same behavior we were getting when we had that conditional on the five lines of code.

So let's walk through what's going on. Right here, you can see the very first part of a ternary operator, is going to be the conditional.

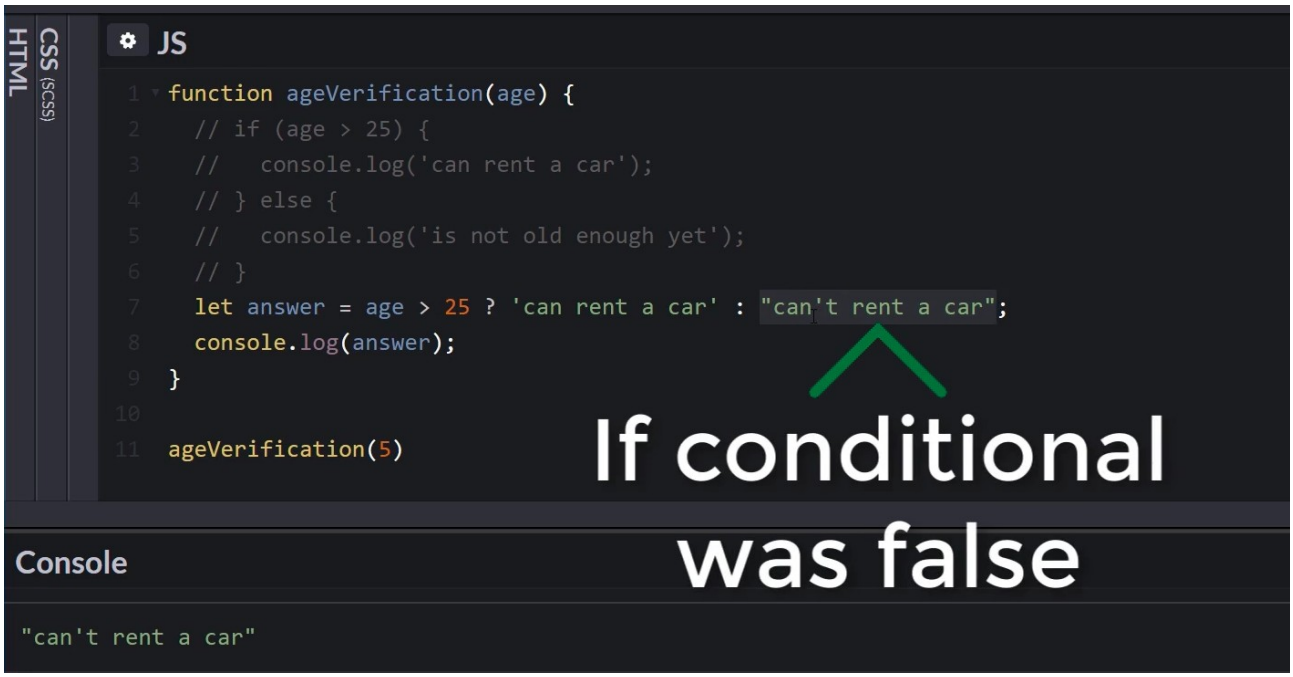


This is exactly the same as saying if age is greater than 25, and so this is the first part, you're going to want to break your ternary operators, the easiest way to think of them is that they're broken into

three parts. The first part is the conditional, the second part, so after the question mark, the second part is going to be, if that conditional is true, I want you to run this code.

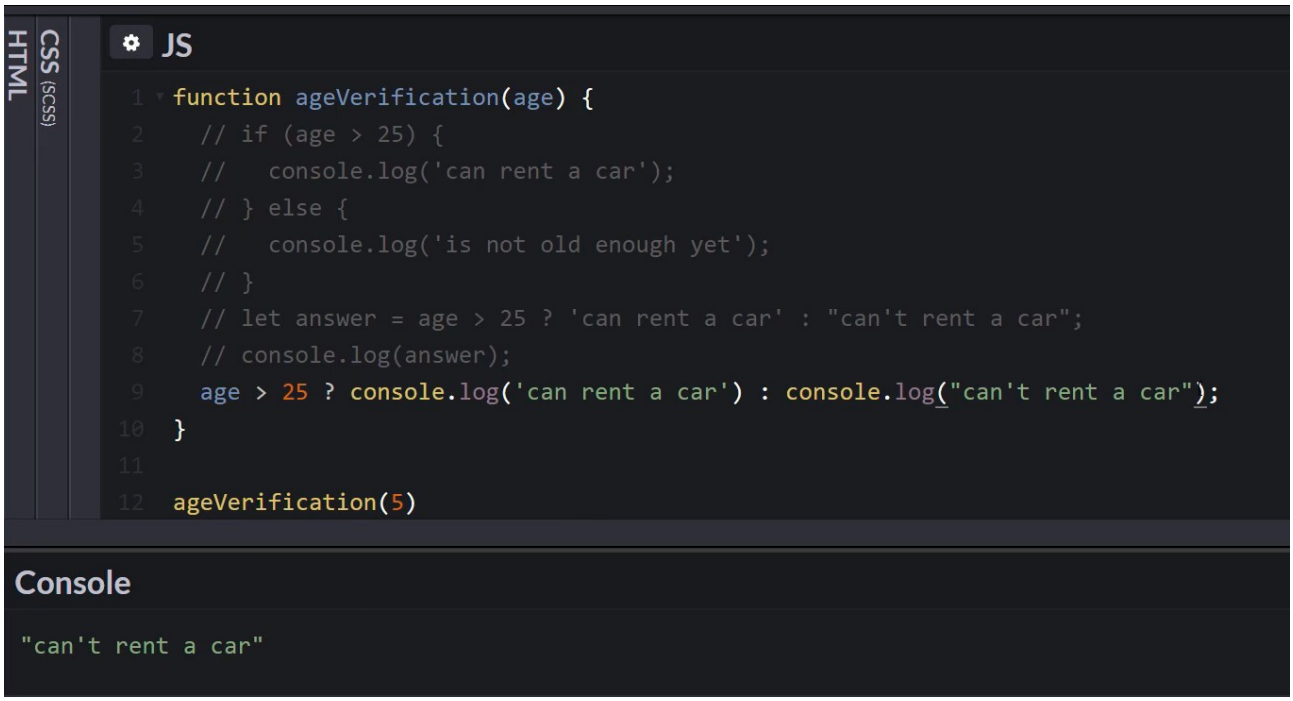


Now if it's not, you have the colon, and now this is going to be what happens if the condition was not met. So if this is false, if the age is not greater than 25, then it's going to skip everything here and then it's going to pass the colon and say, "Okay, we want to return whatever is in this value here."



Now I also could have written out the console log statement here, if you're curious about why I stored this in a variable it's really just for the sake of space because the ternary operators can get a little bit longer. So I could have done something like this.

I could have just said age is greater than 25, then I could have said console log, can rent a car. Then here, console log again and then can't rent a car, just like this. Now if I save this, clear it and run it, then you'll see that we get the exact same behavior where it says, can't rent a car.



But typically, because of the way that this works, whenever you have ... And let me get rid of all of that, just so it's out of the way. So whenever you have a situation where you're using a ternary operator, typically you do not want to put your console log statements actually in the true or the false values here.

So let's walk through the mapping just to make this clear because I cannot tell you how ... don't worry if this looks weird. I can't tell you how many times a student has come up to me and says, I do not like using ternary operators, they don't make any sense they look weird. I can tell you it just takes practice and what helped me the most when I was learning them is understanding what the mapping was.

Remember, the very first part of it is the first part of the conditional. And so we have, if age is greater than 25, we have age is greater than 25. They don't have the if here, but you can just imagine that the if is right in front of it. Then the question mark means that we're now going to break into whatever happens when this is true and when it's false.

This right here is the true part. This is like dropping here into line three where it says, "Can rent a car". Then after this, this little colon here, you can imagine that this is the else. So this is exactly what we have here on line four where it says else and then it says, can't rent a car. It's exactly what we have here where it says is not old enough.

This is the basic way of implementing a ternary operator. Like, you can see if you ever wanted to implement this in a React application or a Vue app, you couldn't write the code like this. If you wanted to put it on one line directly into the HTML, then you're going to have to write it like we have right here.

```
age > 25 ? 'can rent a car' : "can't rent a car";
```

So that's why it's important to know. I'm going to get rid of all of this and now let's get into a little bit more of an advanced example. Before we get into this, I want to add the caveat that what I'm about to show you is important to understand but I would not recommend using it on a regular basis because the single ternary after you practiced it enough it will actually start to become very familiar to you and it's not going to look as weird as it may look the very first time or second time that you've seen it.

What I'm going to show you now is, how you can implement compound logic. So multiple conditions all into the same ternary operator, and I can tell you this is going to look very weird. I've been doing this for a number of years. And when I see a compound conditional built into a ternary, it still takes me a while to kind of dissect the code to see exactly what's happening. So I wouldn't recommend doing this.

But it is important to understand because I have run into a number of projects that I took over and I worked on, where the developer did do this. And it was important for me to understand what their logic and what their process was, because if I didn't, then I'd be lost in the codebase.

So let's walk through a more advanced example. I'm going to create another function here, I'm going to call it `adminControls`. So we're going to say that the purpose of this function is to either show or hide admin controls. Once again, this is something that is similar to a feature you may build into a React or a Vue or angular application.

So I'll say admin controls, it's going to expect to get a user. Then inside of here, let's follow the same exact process we had before. Let's get it working with a regular if/else conditional, and then we're going to turn that into a full ternary operator. So here, I first want to check to see if the user is logged in.

So I'm going to say, if user and now in JavaScript with a conditional, if you're looking for a true or false value, then you don't have to say, if user is true, you can just say, if user and it will assume you mean if the user is true, if it exists. So I'm going to say if user and then I'm going to drop down inside and I'm going to put another conditional.

I'm going to say if, so we know that we have a user, so we can be confident that we can ask the user if they're an admin. So here I'll say if `user.admin` and the same thing, we're expecting a true or false value if the user is an admin or not. Then inside of here, I'm just going to console log and say, showing admin controls.

Then if they're not an admin, we need to have some logic for that. So I'm going to say, if not, then I want to console log and say you need to be an admin. Then we also need to verify, we need to have a backup for if the user is just a guest user. So maybe the user hasn't signed in, so here I'm going to put another conditional and say else and then we'll put console log and then you need to be logged in, just like that.

```
function adminControls(user) {
    if (user) {
        if (user.admin) {
            console.log('showing admin controls...');
        } else {
            console.log('you need to be an admin');
        }
    } else {
        console.log('you need to be logged in');
    }
}
```

Now let's create some examples, some kind of case studies here. I'm going to say, let and we're going to create three user types. So let user one, this is the full admin, so I'll say they have a name and this one will be Kristine, and it's admin with a value of true, so this is just a basic JavaScript object.

And here I will call admin controls and I'm going to pass in user one. Now let's save this and let's just verify that this one's working.

```
let userOne = {
    name: 'Kristine',
    admin: true
}
adminControls(userOne);
```

If we run this, we should get the, showing user controls.

Yes, everything here is showing admin controls. That's all working properly. Let's just give ourselves a little bit of space and we're going to create a few more users. This one's going to be user two and then for this one we'll give it a different name and then admin here is going to be false. Then we're going to call it slightly differently, just like this.

```
let userTwo = {
name: 'Tiffany',
admin: false
}
adminControls(userTwo);
```

Now if I save and I run this, we should have the, you need to be an admin.

So we still have our showing admin controls and now it says, "You need to be an admin." So everything there is working perfectly, they dropped into this user because it was true. But then they were not an admin, so it fell into the else statement, so everything there is working.

And the last one, let's say user three, and this is our guest user. So this is going to be a user who doesn't have any values whatsoever. So we're just gonna say user three and we're just going to say that they're, null. Now, this user three should return that you need to be logged in. Let me clear this, hit run and it should say, "You need to be logged in." Which is perfect.

Everything here is working and our conditional is working. But let's imagine that you need to put this in a ternary operator. Now, this is going to look very weird. Once again, I would not recommend doing this at all, just because I think it leads to unreadable code. But if you ever come across a ternary operator that looks like this, you're going to know exactly what it's doing.

I'm going to create a variable that we're going to store this. So I'm going to say let, response equal and then here we are looking ... Let me comment all this out, just so you don't get a false read on it. So we have this user argument, I'm going to say, user, then from there, we're going to start off the same way we did with our basic examples.

It's going to say, user, question marks where we're going to first check, is the user true. Then this is where it gets weird. This is where we're going to place another ternary operator inside of the very first statement. Because the way that this logic works is it's going to check to see, is this the case? Is this user, do they exist?

So it's the same thing as saying user true and then we're going to drop into what happens if it's true? Well, what happens when it's true in this example? Well, we drop into this second conditional, so that's exactly what we do here with the ternary operator. We're going to say, user.admin and then we're going to give another question mark.

Because this is like asking that second question. So we're going to say, user.admin and then here showing admin controls ... and then we're going to give what happens if they're not an admin. You need to be an admin. Now that we have that, now we need to go into that final else. This is what happens if the user didn't exist. Now we'll say, you need to be logged in. We stored all of that in the response. Let's just console log that, so we'll say console log response. Hit Save, clear this and we should get all of the exact same answers.

```
let response = user ? user.admin ? "showing admin controls..." : "You
need to be an admin": "You need to be logged in";
console.log(response);
```

If I hit Run, there you go. You need to be logged in, right above it was you need to be an admin, and then showing admin controls. As you can see, this looks really weird, one thing I will say is unlike the if/else conditional, the spaces and having these carriage returns where you have all of the code on different lines, that may not be allowed in the JSX, so in your React or Vue application. But you actually can have carriage returns and it is valid code.

You could make this look something like this. This is going to look possibly even weirder but the last time I ran into one of these in the wild in a project, they actually had it all on multiple lines, so I wanted to show that to you. I could say something like, user and then question mark, user admin, have the entire conditional right here. Then put this on another line, just like this.

```
let response = user
    ? user.admin ? "showing admin controls..." : "You need to be an
admin"
    : "You need to be logged in";

console.log(response);
```

For some reason, with the way that JavaScript is compiled, this still will work. If I hit Clear, save, and then run this again, everything here still works. So you may see some examples, that look something like this. So if you ever see this syntax know that the developer created a ternary operator, in this case, a compounded one.

Let's walk through what's going on here, kind of line by line. Let's start at the top. So we're checking to see if the user exists. That's the same thing as just placing the user right there. Same thing as saying, user === true. If that is true, they will drop down into this next line here. And if you prefer, if this is messing you up too much, then let me put this all on a single line again, just so we can read it all from left to right.

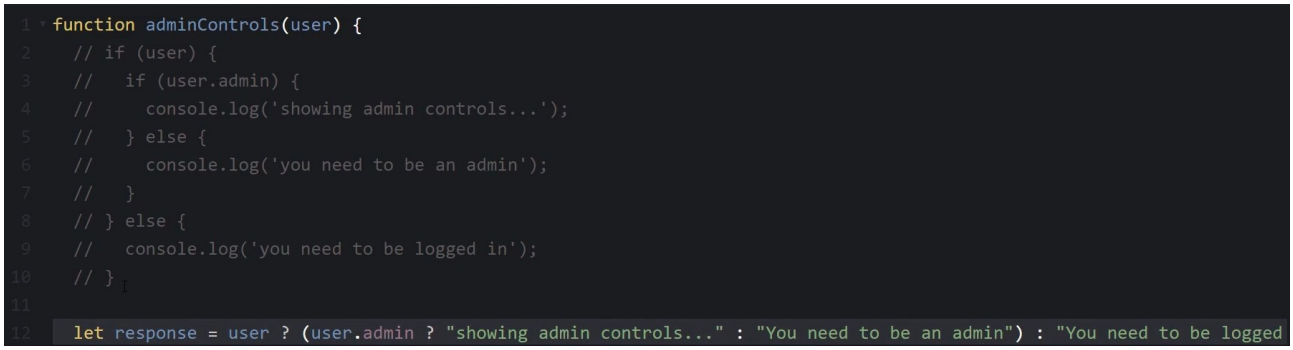
So right here, what we're doing is we're saying is the user, do they exist? Yes. Okay. Well, now it's time to drop into another conditional. One thing that does help me whenever I'm working with this kind of code, is I like to wrap the separate ones up in parens, just like this, this makes it a little bit easier and as you'll see, this also works exactly the same way.

```
let response = user ? (user.admin ? "showing admin controls..." : "You
need to be an admin") : "You need to be logged in";

console.log(response);
```

Now if I hit run, everything still works but I think at least, in my opinion, this is a little bit easier to read and it shows that this is a nested conditional. Technically, you could keep on nesting them, you could have another conditional here, or you could have it in the else block. But to me, even having two of them is honestly a little bit too much. But it's your world, you get to live in it, so build your conditionals however you want. But I just hope I wouldn't have to take over your codebase if that is what you decide on doing.

So what we're just getting back to, we have the user first conditional. They drop into this conditional, this is the same thing as what we have here on lines 3 through 7. Where it says, "If this is the case, I want you to show the admin controls. If not, I want you to ... You need to be an admin." so same process.

Then finally, if the user didn't exist. If this was false, then it skips everything here until it finds the final colon and then it says you need to be logged in. So what we wrote here on line 12 is exactly the same as what we wrote on line two to 10. As you could see the behavior is identical but the difference is, if you ever need to write your conditional all on one line, then this is the syntax that will allow you to do that.



Now if this was confusing to you at all, and do not feel bad if it was, ternary operators are one of the more confusing parts of learning JavaScript, especially in the beginning. Then what my recommendation would be, is to go through the show notes. I'll provide all of this code for you. Put it into your own code pen, use it on your local system, and then play with it.

Make some changes, look and see what happens if you change something in this part of the conditional and see how it maps to what you have here in the normal if/else statement. Go through it until it starts to really sink in and make sense. Then once you get into learning about React and these other frameworks, and you see one of these ternary operators you're going to know exactly what to do.

Code

```
function ageVerification(age) {
  // if (age > 25) {
  //   console.log('can rent a car');
  // } else {
  //   console.log("can't rent a car");
  // }

  return age > 25 ? "can rent a car" : "can't rent a car";
}

ageVerification(30); //?
ageVerification(10); //?

function adminControls(user) {
```

```
  // if (user) {
  //   if (user.admin) {
  //     return 'showing admin controls...';
  //   } else {
  //     return 'You need to be an admin';
  //   }
  // } else {
  //   return 'You need to be logged in';
  // }

  return user
    ? user.admin ? "showing admin controls" : "You need to be an
admin"
    : "you need to be logged in";
}

const userOne = {
  name: "Kristine",
  admin: true
};

adminControls(userOne); //?

const userTwo = null;

adminControls(userTwo); //?

const userThree = {
  name: "Tiffany",
  admin: false
};

adminControls(userThree); //?
```

MODULE 3.

FUNCTIONS

3.1 Section Introduction: Introduction to JavaScript Functions

In this section the course, we are going to dive into JavaScript functions

Now, `functions` in JavaScript are one of the most critical topics related to learning how JavaScript works and how you can build full-fledged programs. If you've never heard of a function and you're new to programming, the way a function works is its kind of like a machine.

This machine has the ability, you can think of a machine on an assembly line or some kind of plant, this machine has the ability to take information in and then it performs all kinds of processes. These are processes that you tell it to perform and then it returns a different value.

So it takes in some kind of value and then it returns a different value. If you can understand that cor conept related to functions, it's going to help you as you learn JavaScript as a whole. I once had a teacher tell me that if you can understand an input and an output for a program, you can understand anything. That is definitely the case when it comes to functions.

I think one of the best ways, before we get into the code, of understanding how functions work is to look at a real-life example. Let's imagine that you're building out a website that has `authentication`, which means that it's like pretty much every Web site that you go to like Facebook or Twitter where you have to type in your email address and then your password.

Then if you typed in the right combination: they'll let you in, let you see your profile, and go through the application. Well if you were to build that out. Imagine a scenario where you need to build that out from scratch.

There are a lot of processes that have to take place: you have to connect to a database, you have to be able to read data in, so you have to read data in from the form, then you have to perform a number of conditionals to make sure that the email address is correct.

That it's in the database, that it matches the password, and then from there you have to redirect the user. If they entered the right information to their profile page, and if they entered the wrong information you have to let them know that they typed in the wrong username or password.

Well, that's a lot of steps, and so what a function can do is it can wrap all those up into a single line of code. What you can do is imagine if you had all those steps, say it's a dozen steps you needed to authenticate a user, if you needed to log users in on multiple pages. If you didn't have functions you'd have to copy and paste that code into every single file that you wanted to authenticate the user in.

That is really a bad practice because then imagine a scenario where you have to change one of those lines of code. You'd have to remember every single place that you entered that code in and then

you'd have to go and make that change. That's very error-prone, and you really would not have a good time programming following that process.

What a function allows you to do is you can take all of that code, each one of those steps that were involved for logging in the user, place them inside of a single function, and then anywhere else in the program that you want to see if a user can log in or not: you simply call that one function.

You pass in the email address, the password, and then the function then is run automatically every time that you call it. I know that may seem a little abstract if you've never worked with it before, but I think it helps to have a real-world scenario on when you'd want to use a function.

Now that you have a high-level idea on how it works. Let's get into the code and let's start building out our own functions in JavaScript.

3.2 Basic Syntax for Building Functions

This introductory guide walks through how to write JavaScript functions. Additionally, we examine the difference between console output and returning values.

THEORY:

What are Functions?

Functions are reusable blocks of code that perform specific tasks. They are essential for organizing and structuring your JavaScript code.

Basic Syntax

```
function functionName(parameters) {
  // Code to be executed
}
```

- **`function` keyword:** Indicates that you are defining a function.
- `functionName`**:** A descriptive name for your function (e.g., `calculateArea`).
- `parameters`**:** Optional input values for the function (e.g., `width`, `height`).
- **Curly braces `{}`:** Enclose the code block that will be executed when the function is called.

Example: A Simple Function

```
function greet() {
  console.log('Hello, world!');
}

greet(); // Calling the function
```

Console Output vs. Returning Values

- **`console.log()`:** Prints output to the browser's console. This is useful for debugging and displaying information, but the output cannot be used directly in your code.
- `return` **statement:** Specifies the value that the function should "return" when it is called. This value can be stored in a variable or used in further calculations.

Example: Returning a Value

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

let sum = add(5, 3); // sum now holds the value 8
console.log(sum); // Output: 8
```

Why is Returning Values Important?

- **Code Reusability:** Functions that return values can be used as building blocks in other parts of your code.
- **Data Flow:** Returning values allows functions to pass data back to the caller, enabling more complex logic.
- **Modularity:** Functions with clear inputs and outputs promote modular code design.

Important Notes:

- Functions are reusable blocks of code that perform tasks.
- `console.log()` prints output to the console, while `return` sends a value back to the caller.
- Returning values is crucial for code reusability, data flow, and modularity.

VIDEO:

I’m really excited to get into this section on functions because when it comes down to it javascript is really built on functions and one of the things that makes javascript so much different than so many other programming languages is how it allows you to work with functions and pass them around, work with them in a much more flexible much more scalable manner than other programming languages.

That doesn't have to make sense right now. We're going to get into that into much more detail throughout this section. This is going to be much longer than say the conditional section just because there is so much to learn about how to use functions in Javascript.

I have the javascript console open right now and we're going to start off by just building a very basic function. The syntax for this is just going to be, say the word function followed by whatever you want to call it. I'm going to say function and then hi thereafter that you need to put parentheses. If you have a function that does not take any arguments then you just put empty parentheses. We'll get into what arguments are later on.

For right now just know that they are other items other components objects are variables things like that you can pass into the function when you call it. So that's the first part. After that, you're going to place curly brackets{}. And so if you're working in the console it's going to put you on the next line and then eventually when we're done we're going to close out the curly brackets.

But right now we're just going to put a very basic console log statement in here. So it's going to say console log. And It'll say hi there and let's close it out with curly brackets and then it's done.

```
function hiThere(){
console.log('Hi there');
}
```

That is your very first function inside of javascript. Now you notice nothing happened. So right there it says undefined what that means is nothing got returned. Nothing happened by just defining how the functions going to be. In order to have this print out what we have to do is say hi there. Put in the parens and then a semicolon hit return. And now you can see that it printed it out. Hi there. So that is how you can call a function in javascript.

Now I want to point something out that is incredibly important, and if you are new to programming then this may take a little while until it really seeps in and you understand exactly what's going on here. And this is the reason why I'm using the javascript console right now instead of using code Penn. because it shows what gets returned. I mentioned earlier on in the course we are going to come back and we're going to talk about what it means to return a value and now it's time to do that.

Our function here didn't actually return anything. All it did it was it printed out to the console, Hi there. That's all well and good. However, there are very few times where you're going to be building a function that does not return anything whatsoever. And the name for that is called a void function.

What that means is this function is going to run but it's not going to return anything back. That would be would be something like where you're running a process. But the process doesn't return anything like starting up. Maybe a portion of a server or something like that. And so for right now you don't have to worry about what exactly it's doing here. So when I say function 'Hi there' console log all that's happening is it's printing to the console.

Now if you want to have this function actually do something so that you could call the function from later on in the program and have the value of hi there returned. That is where we get into what it takes to return a value so I'm going to create a new function. It's going to be hiThereTwo. It's not going to take any arguments and inside of it. Instead of running console.log, I'm going to say return 'Hi there again'. You end the curly brackets to end that function and that's it. Now nothing still is returned yet. Whenever you define a method you're simply defining it nothing is it going to happen when you're finished. Now in order to get that to happen, we need to call it. So we're going to call hiThereTwo. And then semicolon hit return and that is where it's different.

```
function hiThereTwo() {
    return 'Hi there again';
}

hiThereTwo(); // "Hi there again"
```

If you look at both of these when we called hi there, just the first one. It printed this out to the console but then it didn't return anything. When it says undefined that means that nothing got returned. In other words, you say that you wanted to call this and store the value of this inside of a variable. You would not be able to do that because right here doesn't return anything it would simply be called, it print's out to the console, and then it would be done.

However, if you want to actually use the value of the function later on you need to return it. So that is the reason why I chose to use the javascript console for this because it shows if something was undefined meaning nothing got returned by running that code or if it did return a value. I said it a

few times, and the reason I'm saying it is because this part can lead to a lot of confusion, especially for new developers. In understanding what this means, so let's go on with one other example.

I'm going to create a new variable. And this variable is going to be stored in text. And from here I'm going to call our function. So you say hi there. Just the first one hit return. You can see hi there did get printed out. So that got printed out to the console.

```
var storedText = hiThere(); // Hi there
```

But now if I call stored text the variable you can see nothing is inside of it.

```
storedText; // undefined
```

So there is no value that got placed inside of the variable.

So let's try this again now with another variable

```
var storedTextTwo = hiThereTwo();
storedTextTwo; // "Hi there again"
```

Make sure you put the parentheses at the end in order to call it. Now nothing got printed out. But if I call the variable and hit return you can see that now it actually printed that value out because it was returned which means that it actually got stored inside of here. This is going to be a pattern that you follow for pretty much all of your javascript development.

Say that you're building some type of database query you're going to have a function that goes and queries the database and then you're going to store it you're going to return all of those records from the database query store it inside of a variable so that it can be printed out and shown on the page. That is how the return works it's how you're able to pass data and return data or return messages or return all of these different types of components when a function is called.

I hope that made some sense and you now know how to build functions in javascript. Because this is a little bit more weighty of a topic. I'd definitely recommend before going on to the next video that you start to examine how you can build your own functions start to combine some of the other things that we discussed such as conditionals or anything like that put them inside and you have to get everything working but definitely try out how you can do it how you can call the functions store them in variables pass them around. Because this is going to be something that is going to help you out a lot as you continue your development journey.

```
function hiThere () {
    console.log('Hi there');
}

hiThere(); // Hi there

function hiThereTwo() {
    return 'Hi there again';
}

hiThereTwo(); // "Hi there again"
var storedText = hiThere(); // Hi there
storedText; // undefined
var storedTextTwo = hiThereTwo();
storedTextTwo; // "Hi there again"
```

Coding Exercise

Create a function called `greeting` that returns a string when the function is called.

3.3 Variable Scope

This lesson examines JavaScript scope. Specifically, it walks through the differences between local and global scope for JavaScript variables, along with discussing the best practices associated with both options.

THEORY:

What is Scope?

Scope determines the accessibility (visibility) of variables in JavaScript.

- **Global Scope:** Variables declared outside of any function have global scope. They can be accessed from anywhere in your code.

- **Local Scope:** Variables declared inside a function have local scope. They can only be accessed from within that function.

Example: Global vs. Local Scope

```
let globalVar = "I'm global";

function myFunction() {
  let localVar = "I'm local";
  console.log(globalVar); // Can access global variables
  console.log(localVar);  // Can access local variables
}

myFunction();
console.log(globalVar); // Can access global variables
console.log(localVar);  // Error: localVar is not defined
```

Why is Scope Important?

- **Organization:** Helps organize code and prevent naming conflicts.
- **Maintainability:** Makes code easier to maintain and debug.
- **Security:** Protects variables from unintended modification.

Best Practices

- **Minimize Global Variables:** Overusing global variables can lead to naming collisions and make code harder to manage. Use local variables whenever possible.
- **Use `var`, `let`, or `const`:** Always declare variables using `var`, `let`, or `const` to ensure proper scoping.
- **Function Scope:** Utilize functions to create local scopes and encapsulate your code.

Example: Function Scope

```
function calculateArea(width, height) {
  let area = width * height; // Local variable
  return area;
}

let result = calculateArea(5, 10); // result is 50
```

Important Note: Accidentally omitting the `var`, `let`, or `const` keyword when declaring a variable inside a function can lead to unintended global variables. Be cautious!

Important:

- Global variables have global scope and are accessible from anywhere.
- Local variables have local scope and are only accessible within their function.
- Minimize the use of global variables to avoid naming conflicts and improve code maintainability.
- Always use `var`, `let`, or `const` to declare variables and control their scope.

VIDEO:

In this guide we're going to talk about a very important topic in javascript development and that is going to be variable scope.

So far throughout this course is pretty much everything that we've done has been with variables that are in what's called the global scope. What that means is that if we had a full program that each one of these variables would actually be made available to all of the different functions modules classes they'd have access to it. And as you're about to see that can lead to some very confusing bugs and is definitely considered an anti-pattern. So it's very important to understand exactly how variable scope works in javascript so that we can utilize it properly, we can organize our code the right way, and also so we're not going to run into weird behavior where we have one value that is available and could accidentally be called or even overridden later on in a program.

So what we're going to do is I'm going to create a variable here and I'm just going to call it user object and it's going to be an object for just a simple user. So say sample@devcamp.com and they'll have a FULL NAME attribute as well. And this will be Kristine Hudgens.

```
var userObj = {
email: 'sample@devcamp.com',
fullName: 'Kristine Hudgens'
}
```

Nothing really new there, I'm using an object instead of just a plain old string or something. So you can get in the practice of using them because you are going to be using objects quite a bit in your javascript development journey.

The next thing is let's build a function, the reason why this specific guide is included in this module is that I couldn't include it in the earlier one where we were just talking about variables because you can't really talk about variable scope in javascript without talking about functions because they are

directly related to each other and functions are really required in order to organize your variables properly.

So we're going to build a function here and we're going to call it dashboard greeting. Essentially what this is going to be is it's going to be a function that says hi to the user when they log into a web site it's a pretty common feature to add to applications. The very first thing we are going to do is console log and say Hi there and then right after that put a dot and then say concat pass in the User object dot fullname. And now if we want to run this we'd just because it's a function we have to call it. So say dashboard greeting and then hit run and right there it says "Hi there, Kristine Hudgins" it took the object and pulled out the full name and then it concatenated it with our greeting.

```
function dashboardGreeting(){
console.log("Hi there, ".concat(userObj.fullName));
}

dashboardGreeting();     // "Hi there, Kristine Hudgens"
```

Everything there worked. Now if you're coming from a different language than already this variable scope might seem a little bit weird because our function had access to the user object variable. And certain languages do not allow that kind of scope and they do it because they're more strict where javascript is much more flexible as you're about to see. So all of that already works. Now let's talk about some of the issues associated with this. I'm going paste in this variable declaration and assignment up here but instead of Kristine, I'm going to change the name. So now that I place this variable declaration inside of the function this is what is called a local variable, it is local to the function. If I hit run you can see it says "Hi there, Tiffany Hudgens".

```
function dashboardGreeting() {
  var userObj = {
    email: 'sample@example.com',
    fullName: 'Tiffany Hudgens'
  }
  console.log("Hi there, ".concat(userObj.fullName));
}

dashboardGreeting();  // "Hi there, Tiffany Hudgens"
```

This performed an override. But now you may be asking, that seems pretty logical. That is exactly what it should be doing because we redeclare it and reassigned it and changed the value. Well here is where it starts to get a little bit tricky. Let's come down here and say console log and we'll go with userObj.fullname.

Now, what do you think is going to happen here? We already declared and assigned our user object variable with these values then we took the same variable or I should say what was named the same. And then inside of the function, we added different values. Now, we can see what we had in the function. And now we're going to be able to see what our user object is actually set to. If I hit run, now we can see that it prints out. Tiffany Hudgins but then it prints out Kristine. So what exactly is going on?

Well, this is where it gets into the local scope. So when we talk about variable scope it gets routed through and it's managed with the function. So in other words the user object here is actually local to this function. So the only values or the only time you can use these values are while you're inside of the function. Everything outside of that is considered out of scope. And this leads me to one of the most important things that I hope you remember from this guide and that is that you need to be incredibly careful when it comes to what's called polluting the global scope. polluting the global variable scope.

The reason for that is right here we've just created a global variable. If we have a program that has hundreds of files we may rename. Or we may name another variable later on user object. So it's a generic enough name we may rename it and then accidentally override that value and then all of a sudden our program breaks what if the new user object doesn't even have a full name function but we just overrode it with something new. That could lead to all kinds of issues, the best practice is to not even have any global variables or to have a very limited number of them because whenever they're in functions you can use them the way they're supposed to without having that weird type of behavior.

Now I want to add one caveat before we finish and that is that I'm going to put that code back. Now there's one very weird thing that if you have never seen it before might seem a little bit tricky. If I get rid of the VAR declaration then you're going to see something kind of odd. I'm going to clear this out. Hit run and look at that. Now our user object is no longer within the scope of the function. This is no longer local. So that is something they have to be very careful on. And it's one of the reasons why it's very important to put VAR or LET whenever you're declaring a variable.

The reason for it is because if you get out of the habit of doing that then there's a very good chance that you might accidentally and even while inside of a function you may accidentally create a global variable. So that's what the syntax is if you want to create a global variable while in a function then you just leave off the VAR keyword and it will simply go create it, the way if you remember back to when we talked about javascript variable hoisting. It'll pull it up and it'll treat it as a regular variable.

That's definitely something that you want to avoid. And so I recommend whenever you're using these functions you want to make sure you keep them local. Make sure you put a VAR there and then try to prevent yourself from using too many global variables because you will run into a number of very confusing bugs. So that is an introduction to variable scope in javascript.

```
var userObj = {
  email: 'sample@example.com',
  fullName: 'Kristine Hudgens'
}

function dashboardGreeting() {
  var userObj = {
    email: 'sample2@example.com',
    fullName: 'Jordan Hudgens'
  }
  console.log("Hi there, ".concat(userObj.fullName));
}

dashboardGreeting();
console.log(userObj.fullName);
```

3.4 Function Expressions & Function Declarations

This lesson introduces function expressions in JavaScript. Additionally, we'll examine the key differences between function expressions and traditional function declarations.

THEORY:

Function Declarations

Function declarations are the traditional way to define functions in JavaScript.

```
function greet() {
  return "Hello!";
}
```

Function Expressions

Function expressions assign a function to a variable. They can be anonymous (without a name) or named.

```
let greet = function() { // Anonymous
  return "Hello!";
};

let greet2 = function greetMe() { // Named
  return "Hello again!";
};
```

Key Differences

- **Hoisting:** Function declarations are hoisted, meaning they can be called before they are defined in the code. Function expressions are not hoisted.
- **Flexibility:** Function expressions can be passed as arguments to other functions or assigned to object properties, providing more flexibility.
- **Use Cases:** Function expressions are often used for closures, callbacks, and creating functions on the fly.

Example: Dynamic Menu Builder

```
let age = 8;

if (age <= 10) {
  let buildMenu = function() {
    return "Kids' Menu";
  };

  console.log(buildMenu()); // Output: Kids' Menu
}
```

This code demonstrates how a function expression can be used to create a function within a conditional block.

Why Use Function Expressions?

- **Dynamic Function Creation:** Create functions on the fly based on conditions or logic.
- **Closures:** Create functions that "remember" their surrounding environment.
- **Callbacks:** Pass functions as arguments to other functions.

Best Practices

- **Use Descriptive Names:** Give your function expressions descriptive names (if named) to improve code clarity.
- **Consider Context:** Choose function declarations for straightforward functions and function expressions for more dynamic scenarios.

Important notes:

- Function expressions provide a flexible way to define functions.
- They are not hoisted, unlike function declarations.
- Use function expressions for closures, callbacks, and dynamic function creation.

VIDEO:

The idea of functions is so important in javascript that there are a number of ways that we can use them.

So far we've talked about building functions using this type of syntax where you say something like function greeting and then inside of it we can return something like Hey there.

```
function greeting(){
return "Hi there!";
}
```

And this all works.

Now, this is what is called a function declaration. But now we also have another option in javascript and it's what is called a function expression. Now you may also hear these referred to as anonymous function or even named anonymous function expressions. Typically I just call them function expressions and that's usually what you'll hear them called in the regular development circles. But now let's talk about what the syntax for that is. Because the flow is a little bit different and there are some very subtle differences between function declarations like we have right here and function expressions.

To start off with an expression a function expression is a function that is stored inside of a variable. So in other words what we would do is actually change this to say var greeting equals so we perform our assignment and then we can just say function. We don't name it. That's where the anonymous part comes in. And here we can just return say Hi there again and this is going to perform pretty much the same way in most cases.

```
var greeting = function () {
return 'Hi there again';
};
```

This is all pretty basic and this is the syntax and actually I'm missing one thing. You're also supposed to have a semi-colon at the end of a function expression or else some of the things like a Linter which checks for code quality would throw an error. So that is a function expression. Now your very first question may be why in the world do we need a second way of writing functions. This seems to be exactly the same as what we have here.

Let's test it out just to make sure. I say console log greeting and then pass this and let's rename this one we'll say greeting to and. Now if I copy this and call greeting two both of these should run.

```
var greetingTwo = function () {
return 'Hi there again';
};

console.log(greeting());
console.log(gretingTwo());

//"Hi there!"
//"Hi there again"
```

And we have. Hi there. And hi there again. So this is all working properly and once again we're getting back to the question on why in the world would I need this.

Well let's get rid of these two and I'll put them in the show notes so that you have access to them.

But now let's go and let's build a program where you can see why this is so important. So I'm going to start off by saying var age and let's set it equal to 3 years old. And now we're going to create a conditional. So we say if age is less than or equal to 10 then I want you to do everything inside of here. So what we're going to do and this is something that's getting us a little bit closer to real-world development. Imagine a scenario where you had a Web site for a restaurant and part of the process was if you check to see if the user was under 10 years old then you're going to go and build a kid's menu and you're going to build a kid's menu for them and that way you would have some dynamic behavior. So I'm going to create a function called build menu and I'm gonna use a function expression. So if I say var build menu and set it equal to the function and then inside of it where you're not going to build the menu obviously I'm just going to say return Kids menu and then put a semicolon at the end of the return statement and one after the last curly bracket.

Now if I console log this out where I call build menu and then make sure you put your parens at the end of it. If I hit clear and run this you can see it says kids menu.

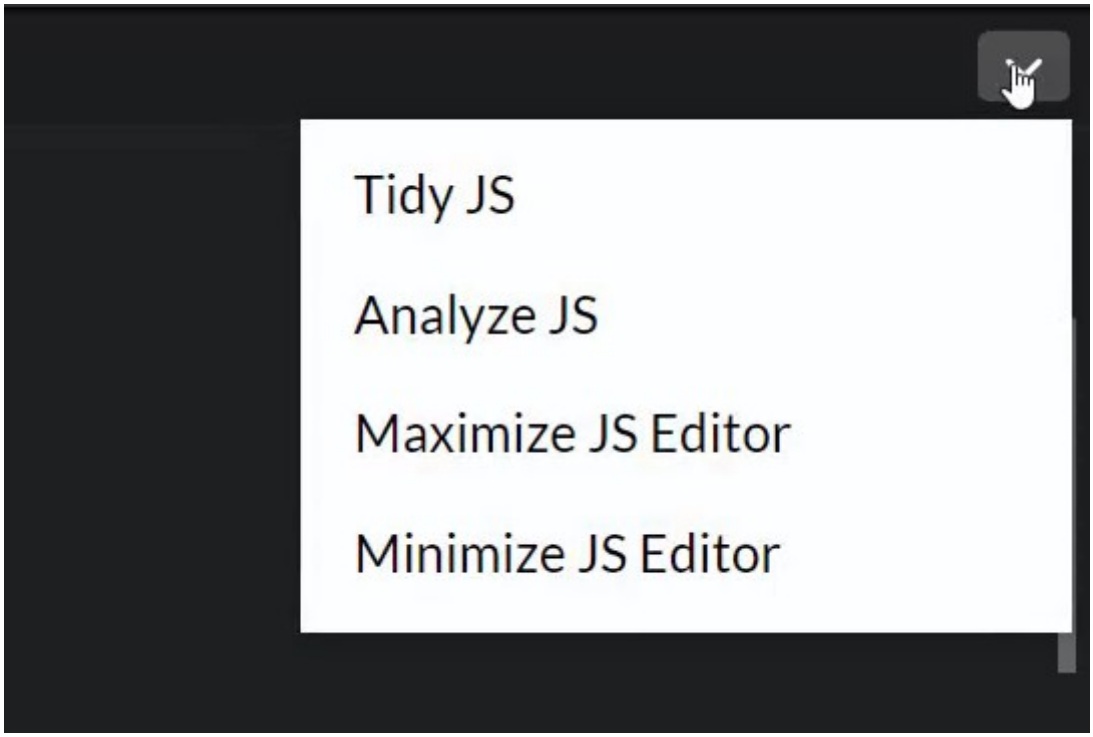
```
var age = 3;

if (age <= 10) {
  var buildMenu = function () {
    return "Kids' Menu";
  };
console.log(buildMenu());
}
//"Kid's menu"
```

Now let's see if we could do the same thing. So what we've done essentially is we have a conditional here and we're saying I only want you to go through this process of building the menu. If the age is less than or equal to 10 imagine this also saying that you have a web application where if a certain condition is met then you want to go run an API query. So you want to call some other server and have other things brought back. Well, you need to build a function in order to do that. So can we build, can we write a traditional function and declaration this way? Well, let's try. If I say buildMenuTwo just with the regular function declaration and inside of it say return and then say another kids menu put a semicolon at the end and now let's copy the console.log and change it to buildMenuTwo.

```
function buildMenuTwo () {
return "Another kids' menu";
}

console.log(buildMenu());
console.log(buildMenuTwo());
}

// "Kid's menu"
//"Another kids' menu"
```

Now, what do you think is going to happen if I hit clear, and hit run? Both of them still come out. So what gives?

Because I told you I was going to show you the difference between the two. Well, the difference is apparent for certain javascript engines. So right here this all seems to work exactly the same. If you remember back to one of the very first videos where I showed you around CodePen when I showed you right here in the top right-hand side where it has a javascript analyzer.



Let's run this and see what it has to say. Well, it liked our function expression it said everything there is perfectly fine moving down here though. Now we have an error where it says the function declarations should not be placed in blocks. And when they mean in blocks they mean inside of curly braces in things like a conditional or a try-catch block or something like that. And that is exactly what we're trying to do we're trying to generate a function on the fly and that is not what function declarations were intended to do that is what a function expression was intended for.

So it says should not be placed in blocks, use and it even tells us right here which is nice. Use a function expression, or move the statement to the top of the outer function. So this is something very important and this is going to start getting having close this out. This is starting to get us closer to building dynamic type of applications because you are going to have many times where you need to build functions on the fly. You may want to store a function inside of an object and all those things are not what you would do with a standard type of function declaration. This is what you would use for a straight right out of the box kind of functions whereas function expressions are more modular.

That's the easiest way to try to remember this is if you have something that you need to be able to move around or a function you need to be able to create at any given point. That is what function expressions are for, whereas a declaration like this is only when you have it outside of that block. So you'd have something like this and then that would work exactly the way that it was intended for but not in how we were trying to do it right here. So that is how you can work with function expressions or anonymous functions in javascript just like we have right here and also a walkthrough on the difference between function expressions and function declarations.

```
var greeting = function () {
 return "Hi there!";
};

var age = 4;

if (age <= 10) {
  var buildMenu = function () {
    return "Kids' Menu";
```

```
  };

  function wrongMenuBuilder () {
    return "Wrong Kids' Menu";
  }

  console.log(buildMenu());
  console.log(wrongMenuBuilder());
}
```

Coding Exercise

Build a function expression called myFunction and have it return true:

```
var myFunction
```

NOTES:

```
/*
FUNCTION DECLARATION:
Hoisting is used when ... you need to declare something for the entire code
Hoisted functions are the regular way to create functions.

function example() {
   code_of_the_hoisted_function_which_will_be_available_at_the_entire_code
}


FUNCTION EXPRESSION:
They can't be called BEFORE being declared, so, not hoisted.
That's why they are used to define a function inside an specific code part, like a loop, or so on.

var data = function () {
   code_of_the_function_expression_that_only_works_here
}
*/
```

3.5 Function Arguments

This guide describes how to utilize function arguments/parameters in javaScript. Additionally, we'll examine how to define default values for function arguments.

THEORY:

What are Function Arguments?

Function arguments allow you to pass data into functions, making them more flexible and reusable.

Example: Full Name Function

```
function fullName(firstName, lastName) {
  return lastName.toUpperCase() + ", " + firstName.toUpperCase();
}

console.log(fullName("John", "Doe")); // Output: DOE, JOHN
```

Explanation

- `firstName` and `lastName` are the arguments (parameters) of the `fullName` function.
- When the function is called with `fullName("John", "Doe")`, the values "John" and "Doe" are passed into the function.
- Inside the function, `firstName` becomes "John" and `lastName` becomes "Doe."

Handling Missing Arguments

If a function expects arguments but they are not provided, JavaScript will assign them the value `undefined`.

Default Parameter Values

You can provide default values for function arguments. If an argument is not provided when the function is called, the default value will be used.

Example: Default Language

```
function fullName(firstName, lastName, language = "English") {
  return lastName.toUpperCase() + ", " + firstName.toUpperCase() + " -
" + language;
}

console.log(fullName("John", "Doe")); // Output: DOE, JOHN - English
console.log(fullName("John", "Doe", "Spanish")); // Output: DOE, JOHN
- Spanish
```

Explanation

- The `language` parameter has a default value of "English."
- If no language is provided, "English" is used.
- If a language is provided, it overrides the default value.

Alternative Default Value Technique

You can also use the logical OR operator (`||`) to set default values.

```
function fullName(firstName, lastName, language) {
  let lang = language || "English"; // If language is undefined, use
"English"
  return lastName.toUpperCase() + ", " + firstName.toUpperCase() + " -
" + lang;
}
```

Why Use Default Values?

- **Flexibility:** Allows functions to work with or without optional arguments.
- **Readability:** Makes code easier to understand by clearly indicating default behavior.
- **Maintainability:** Simplifies code maintenance by reducing the need for conditional checks for missing arguments.

Important notes:

- Function arguments allow you to pass data into functions.
- Default values provide fallback options for missing arguments.
- Use the logical OR operator (`||`) for an alternative way to set default values.

VIDEO:

So far we've covered a number of different characteristics with functions both with declarations and expressions. And now we're going to get into how we can actually pass data into functions with function arguments. Now you may also hear these called function parameters and that's what we're going to go through now.

The very first thing I'm going to create is a function and this is going to be something that I consider a more real-world, mainly because I build this function on a pretty regular basis for the applications I built. So this is going to be called fullName and this is going to take in a firstName and a lastName and then it's going to kind of format them in a way that maybe is considered a little bit more formal. So here is a first name followed by the last name. And these are the function arguments we have. The first parameter right here and then the second one here followed by a comma.

```
function fullName(firstName, lastName) {
}
```

Now the way that this works is when you pass in one of these words you can use it however you want inside of this function block so in between the curly braces you can treat this just like a variable. Or any other kind of data points you can reference it in whatever value you pass in when you call it is what is it's going to represent during the execution of the function.

So here what I'm going to say is return and I'll say lastName and then toUppercase and you don't have to do this one part of this is just to show you a way of formalizing a name and so you can see that you can do more than just call it. You can also perform various functions on top of the data that you're calling.

So here I'm going to say .toUpperCase and then I'm going to concat this with a comma and a space and then I'm going to add the firstName on top of it and also say .toUpperCase for that one as well. And then I'm going to close it off and that is all we need to do.

```
function fullName(firstName, lastName) {
return lastName.toUpperCase() + ", " + firstName.toUpperCase();
}
```

Just to review what we have here is a function declaration with two arguments. And you could also say these are two parameters. The first one is first name. Second one's last name. Now if I ran this, this was going to act kind of weird so if I run the function just by itself. Essentially what this is saying is we just tried to call a function of two uppercase and we weren't able to do that because there was nothing there because it was undefined. So how do we fix this?

Well, this is where we actually pass in real-world values. So here you say Kristine comma and then Hudgins. Now let's try to run this, running this you can see this now work. So it took in the last name it took in both the first name and last name and then we're able to call the last name we made all of the characters uppercase. Then we added a common space and then the first name all uppercase So everything here is working perfectly fine.

One thing that can be a little bit confusing is, let's create another function. So this is going to be a function and we're just going to call it our sample function with our arg1 and arg2. And now if I want to do something like console log arg1 and then console.log arg2. If I call this if I just say sample and don't pass in any arguments you can see that it prints out undefined and then it returns undefined.

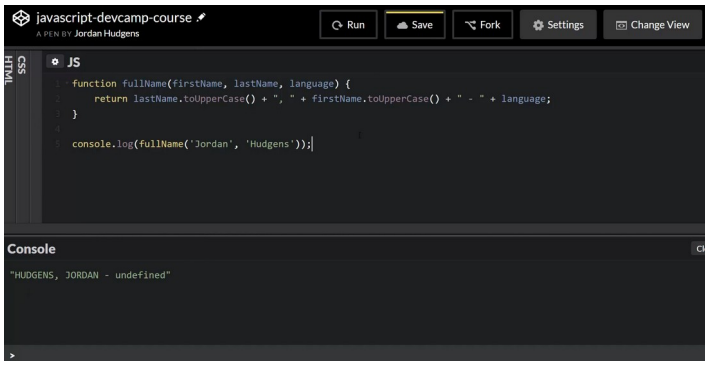
```
function sample(arg1, arg2) {
console.log(arg1);
console.log(arg2);
}

sample();   //undefined
```

So many programming languages would throw an error right here and for a good reason, it's because the function was supposed to have two arguments but nothing got put in. The reason we received an error earlier is that we actually tried to call a function on top of one of those parameters. Now though where we're just passing in and console logging it then javascript does not complain. This is part of one of the issues or one of the complaints people do have with javascript is it is very easy to build a program that does not throw an error but simultaneously doesn't actually work right. And this is a great example of that in languages that are more strict. If you tried to declare a function here and then you tried to call it without the parameters that you required it would throw an error and it would say like the way the programming language Ruby says that you tried to call a method or a function that has multiple arguments but you didn't pass any.

That's pretty common among most programming languages, not javascript though it's very flexible and so it will let you run this which can lead to some confusing types of things. so you need to make sure that you're very careful with how you are calling those, and also how you're declaring them.

I'm going to cover one other task here. Which is how to be able to build something and to have default values because that's a very common thing that you need to perform. Whenever you are building out these types of systems, whenever you're using arguments are many times where you will build a function it'll have, say two arguments inside of it that need to be supplied and then one of them you want to make optional.

How can we do that? Well I'm going to copy our full name function here I'm going to use it but I'm going to add in another parameter so I'm going to say language so we could say this is the first name last name. This is the language that is spoken and we want to have a default argument. In fact, let me put this because it can be a little bit challenging to edit items in the console. Let's go into CodePen. So we have full name, first name, last name, and language. And I want to do something here where we're concatenating languages well. So I can say like a little dash and a space and then concatenate that with just language. Now the way that you can do this is to say that I was to try to console.log the sound, console.log file name, and pass in let's see Jordan Hudgins and nothing else.



If I were to run this now we'd get Jordan Hudgins in an undefined which is not what we're looking for. I want to have something where English is the default language but we can override it if we want to.

The way you can override that here is, I can say language and then set this equal to so we can perform assignment here and we can say language or English.

```
language = language || 'English';
```

And if I run this you can see that this now says English even though we didn't define it. And just to be careful I need to also make this a var. Clear this out and hit run, and you get extra points if you can guess why I wanted to do this. This actually goes back to our conversation on the variable scope. I accidentally when I left var out here I created an accidental global variable which would be a very bad idea. So I created var to make sure that this is specific and local just to the function. And what in the world was I doing here? This may look kind of weird.

If you remember back to when we're talking about compound conditionals we talked about how you can have multiple conditionals by using two ampersands in a row that is essentially the programming way of saying. And well when you have two pipes(| |) in a row and in case you're wondering where the pipe is on your keyboard if you've never used it it is the key right under delete or backspace and right of return or enter. So this if you put two of these together what javascript interprets this as is the word or.

What this is going to do, just to follow it from a data flow perspective. We have the full name with first name last name and the language. Now var language equals language or English. And the way that this works is the javascript engine looks at it and it checks to see is the language defined. If it is it's just going to put that inside and it's going to skip this entirely. However, if it finds this to be undefined then it says well let's look at the right-hand side. So either use this or use this. And if it finds something here then it will store that inside of language. So that is one of the common ways of defining a default value. And so now if I come here and instead I want to change this to be something like Spanish if I clear this out and run this now it gets overridden and it's Spanish because here what happened is when it found the argument of language it stored it and then it skipped over everything here and it just stored it inside our language variable which is what we are returning. So that is how the arguments work inside of javascript.

```
function fullName(firstName, lastName, language) {
    var language = language || 'English';
    return lastName.toUpperCase() + ", " + firstName.toUpperCase() + "
- " + language;
}

console.log(fullName('Jordan', 'Hudgens', 'Spanish'));
```

Coding Exercise

Create a function called sum that adds two arguments together and returns the sum of the two numbers.

NOTES

```
/*

function functionName(its, arguments, here, variables, are, passed,
to, arguments) {
    code_of_the_function
}

*/
```

3.6 Function Arguments: Reference vs Value

This guide discusses the key differences between passing arguments via reference and passing them by value

THEORY:

Reference vs. Value

- **Pass by Value:** When you pass a primitive data type (like a number, string, or boolean) to a function, a copy of the value is created. Changes made to the parameter inside the function do not affect the original variable.
- **Pass by Reference:** When you pass an object (including arrays) to a function, a reference to the original object is passed. Changes made to the parameter inside the function will affect the original object.

Example: Objects (Pass by Reference)

```
let user = { name: "John" };

function changeName(userObj) {
  userObj.name = "Jane";
}

changeName(user);
console.log(user.name); // Output: Jane (the original object is
modified)
```

Example: Variables (Pass by Value)

```
let num = 10;

function changeNum(number) {
  number = 20;
}

changeNum(num);
console.log(num); // Output: 10 (the original variable remains
unchanged)
```

Why the Difference Matters

Understanding how JavaScript handles arguments is crucial for:

- **Avoiding Unexpected Side Effects:** Modifying objects passed by reference can lead to unintended consequences if not handled carefully.
- **Data Integrity:** Ensuring that your original data remains unchanged when you don't intend for it to be modified.
- **Debugging:** Makes it easier to track down bugs related to unexpected data changes.

Working with Objects

If you need to pass an object to a function but don't want to modify the original object, you can:

- **Pass a copy of the object:** Create a copy of the object and pass the copy to the function.
- **Pass only the necessary properties:** Instead of passing the entire object, pass only the specific properties you need.

Example: Passing a Property

```
let user = { name: "John" };

function changeName(userName) {
  userName = "Jane";
}

changeName(user.name);
console.log(user.name); // Output: John (the original object is not
modified)
```

Important:

- Primitive data types are passed by value, while objects are passed by reference.
- Modifying objects passed by reference can affect the original object.
- Be mindful of how you pass arguments to avoid unintended side effects.

VIDEO:

Now that we've discussed what parameters are and how function arguments work. I'd be remiss if I didn't go into detail and talk about the types of items you can pass into javascript functions. So if we pass in an object you may be surprised to learn that it behaves much differently than if you passed in a variable. And the reason for this is because we have the concept of `reference vs value` and this can be very confusing especially if you've never seen it before. But essentially the way it works is when it comes to arguments in Javascript functions you are going to pass in objects and those are going to get treated by reference which means they're not actually grabbing the values they're grabbing a reference to the original object.

Whereas if you pass in a variable it is simply going to essentially make a copy of that variable or of that variable's value and it's going to use that. This can have some pretty far-reaching implications if you've never seen it before so I'm going to start off by creating an object here. I'm going to say some user and give it just an attribute of a name so I'm going to say Jordan and that is our someUser.

```
var someUser = {
name: 'Jordan'
}
```

Now let's create a function. So function and I'm going to make a function declaration here called nameFormatter. It's going to take in a user which we're going to pass a user object in and it is going to return the user.name, but let's imagine that we did something to change the value of the name.

```
function nameFormatter (user) {
    return user.name = 'Oops';
}
```

So if we did that everything is going to work.

But we're going to have a little bit of some weird behavior. So if I say nameFormatter and I'm going to pass in some user as the argument it returns "Oops" which is pretty much what you'd expect that it would return.

```
nameFormatter(someUser); // "Oops"
```

But now what happens if we pass in some user?

```
someUser; // Object {name: "Oops"}
```

Now we can see that some user the object has its name and it's changed to Oops. That is probably not what we're going for when we're passing that value in.

Now let's see how it works with variables. Let's create a variable so I'm going to say var and just say some name equals Tiffany.

```
var someName = 'Tiffany';
```

Now we can create a function again. So when you say function and someOtherNameFormatter and this is just going to take in a name it's not going to take in the full object. And here if I say return and then name equals Oops and closeout that block, everything seems like it works. And now if I called someOtherNameFormatter and pass in our variable it prints out Oops.

```
function someOtherNameFormatter(name) {
    return name = 'Oops';
}

someOtherNameFormatter(someName); // "Oops"
```

So far so good. Everything seems like it's working the exact same way.

But now if I look for someName again you can see it's still equal to Tiffany. So this is the main difference. When we pass an object in as an argument it is going to be treated as a reference which means that the function is going to go and look at the reference to the object.

So in this case when we called some user it got redirected up to someUser and it said Ok someUser has a name attribute. And now we're going to change it. Given our function declaration rules, we're going to change that user's name to Oops. And that is part of what we're doing we're doing more than just returning this. We're also performing all the tasks inside of it. And so it went and had a side effect. And this is what a visual side effect is called is where you run a function and the function does more work than you thought it was going to do. It performs other tasks than simply what you were intending for it to do and that can lead to some very weird, very confusing types of bugs.

Now if we come down and we come to our second variable where we have someName and when we call another function and notice this is doing the exact same thing the only difference is it's not

an object so we're not using the dot notation but essentially we're grabbing the whatever we've passed in and then we're setting it equal to something else when we call this function, we get are something else, we get our Oops and that's what we're looking to do. Then if you call the original value that got passed in right here you look for that argument. It did not go change the value of the variable. That is the key difference between something that is passed by value and something that is passed by reference.

Now one of the first questions you may have is, what happens when I need to pass in an object? How can I do that?

One of the most logical ways that I've seen to do that is to actually just pass in the actual value itself. I'm going to copy this function so you can see everything is exactly the same. And here I am going to say userName just so we can be more explicit about it and instead of calling and traversing using the dot notation here. I'm going to hit return and I'm going to create another object because this one's already been changed here or let me just change it so we can see that someName or I should say someUser.name

```
function nameFormatter (userName) {
return userName = 'Oops';
}
someUser.name; // "Oops"
```

You can see right now it's Oops. Let's fix that. Remember we can change an object value by simply calling it just like this. And now if you call some userName again now you can see it's been updated. So what we can do is with our nameFormatter the way that we've changed it up. If I call nameFormatter, now and instead of passing in the whole object I just pass in someUser.name then let's see if this is any different. So I run this. It still prints out Oops everything is exactly the way we'd expect.

Now if I say someUser.name now, you can see that it has not changed the value and that is how you can pass in an object without having its reference points change. In other words without having that weird side effect where the value in the object got updated and now you can treat the parameters and the attributes inside of your objects the same way that you'd treat a variable.

```
var someUser = {
    name: 'Jordan'
}

function nameFormatter (user) {
    return user.name = 'Oops';
}

nameFormatter(someUser); // "Oops"

someUser; // Object {name: "Oops"}

var someName = 'Tiffany';

function someOtherNameFormatter(name) {
    return name = 'Oops';
}

someOtherNameFormatter(someName); // "Oops"

someName; // "Tiffany"

function nameFormatter (userName) {
    return userName = 'Oops';
}
```

```
someUser.name; // "Oops"

someUser.name = 'Kristine';

nameFormatter(someUser.name); // "Oops"

someUser.name; // "Kristine"
```

Coding Exercise

Overwrite the `someUser.name` value so that it says "Jordan" instead of "Blank".

```
var someUser = {
    name: 'Blank'
};

function changeName(user) {
    return // write the code to overwrite someUser.name
}

changeName(someUser);
```

3.7 Closures

This lesson examines how to work with JavaScript closures. This will include walking through how to return closures from a function.

THEORY:

What are Closures?

Closures are functions that have access to variables from their outer (enclosing) function, even after the outer function has finished executing. This 1 allows closures to "remember" their environment and maintain state.

Example: Batting Average Calculator

```
function battingAverage() {
  let hits = 100;
  let atBats = 300;

  return {
    getCurrentAverage: function() {
      return hits / atBats;
    },
    updateHitsAndAtBats: function(hit, atBat) {
      hits += hit;
      atBats += atBat;
    }
  };
}

let altuve = battingAverage();
console.log(altuve.getCurrentAverage()); // Output: 0.3333333333333333
altuve.updateHitsAndAtBats(0, 20);
console.log(altuve.getCurrentAverage()); // Output: 0.3125
```

Explanation

- The `battingAverage` function returns an object containing two closures: `getCurrentAverage` and `updateHitsAndAtBats`.
- These closures have access to the `hits` and `atBats` variables even after `battingAverage` has finished executing.
- The `altuve` variable holds an instance of the `battingAverage` closure, maintaining its own state of `hits` and `atBats`.

Benefits of Closures

- **State Management:** Closures can maintain state across multiple function calls.
- **Encapsulation:** Closures can hide data and functionality, promoting modularity and preventing naming conflicts.
- **Dynamic Behavior:** Closures can create functions that adapt their behavior based on their environment.

How Closures Work

When a closure is created, it forms a closed environment that includes:

- The closure function itself.
- The variables in the surrounding scope (lexical environment) where the closure was created.

Real-World Applications

Closures are used in various scenarios, such as:

- **Callbacks:** Functions passed as arguments to other functions.
- **Event Handlers:** Functions that respond to user interactions.
- **Data Privacy:** Creating private variables and methods within objects.

Important Notes:

- Closures are functions that "remember" their environment and maintain state.
- They are created when a function references variables from its outer scope.
- Closures are powerful tools for state management, encapsulation, and dynamic behavior.

VIDEO:

This is going to be a really fun episode. In this guide, we're going to build more of a real-life type program. That is going to calculate a batting average and we're going to learn about javascript closures in order to do it.
I'm going to create a function declaration here. I'm going to say function and then battingAverage. And this is going to take no arguments. And now inside of it, I'm going to create some local variables. One thing I like about the program we're going to build. You may notice we're going to include a number of the things that we learned in the past few sections all together here so we're going to talk about local variables talk about functions how items get returned. We're going to talk about nested functions which are, in this case, are going to be closures and then we're even in to perform some computation.

The first thing we're going to do is set hits a local variable and set this equal to a hundred just to start off. And now var atBats set this equal to 300.

```
function battingAverage () {
```

```
    var hits = 100;
    var atBats = 300;

}
```

Now any normal program these would probably be calling a database, getting that count, and then that's how these are going to be updated.

Now normally up to this time what we've done is we have simply returned a value. So we've taken something like hits and atBats in this case and then we return to some type of computation on those.

Now I'm going to talk about what closures are and how we can use them in order to give a more dynamic type of feel to our programs. So essentially what a closure is, is it's being able to wrap up an entire set of behavior and usually wrapped up in a function and be able to use that, pass it around, and call it however you need to.

What we're going to do is we're going to create two closures. We're going to create one for getting the current average and then we're going to get one that is going to be updating. One is simply returning a value back kind of like querying a database. The other one is actually going to perform some action that's going to set some values and then we can call our get current average after that one.

So I'm going to have our return statement here. Return a javascript object this is going to return an object and the object is going to contain two closures. Now one thing to keep in note, just a nomenclature kind of topic. Whenever you have an object and you have functions declared or defined inside of that javascript object those functions are technically called methods. So usually when you're going to see something like this, the proper name for them is a javascript method and our method name is going to be getCurrentAverage and just like a normal key-value pair that's going to just say getCurrentAverage and then we're going to use an anonymous function here. So I'm going to say function and then inside of this function we're just going to return our hits which is our local variable up there divided by atBats which is how you'd get an average of something.

```
return {
   getCurrentAverage: function () {
     return (hits/atBats);
   }
```

So the math isn't too crazy we're just taking our hits dividing it by atBats and that's all we need to do. So that is the first part.

Now that is one function or one method. And now we're going to have another one and this is going to be named updateHitsAndAtBats, this one same thing it's going to have an anonymous function that's assigned to it and we'll say function hit is going to be one argument and atBat is going to be another argument and notice that even though this is an anonymous function we can pass arguments to it the same way that we would before. And this is the reason why I wanted to talk about function expressions before so that you'd be able to see how they could work as closures. So we're going to perform two tasks here first we're going to take our hits variable our local variable. We're going to increment by one or however many we want to pass in, and technically you could pass in more than one, you could pass in 10 hits if they got 10 hits and you want to update the system. Then we're going to do the same thing with atBat. So we're going to say atBats and we will increment that by atBat.

```
   updateHitsAndAtBats: function (hit, atBat) {
     hits += hit;
```

```
    atBats += atBat;
  }
```

So that's a little bit more work than usually, we've been doing each lecture. So let's kind of review it.

We have declared a function called battingAverage. It takes no arguments and we have started it off with two variables and these are local variables local to battingAverage if we call them outside of it, it would not work.

Now our function returns two methods. It returns a getCurrentAverage which simply is going to grab the hits, divide it by the atBats and that is going to be the current average and then we have an updater action here, we have updateHitsAndAtBats which is another method it takes two arguments a number of hits, a number of atBats. And then increments both of those. So let's see if this works I'm going to create a new instance of this batting average say var altuve, the baseball player and say battingAverage and just create it just like that. Now nothing's going to happen here.

But now let's see what we can do, we can call console.log(altuve.getCurrentAverage());

Now let's see what we have. So if I save this and if I run this you can see that this worked. It gave us a batting average of 333 which is accurate if you divide 100 by 300.

```
var altuve = battingAverage();
console.log(altuve.getCurrentAverage());
// 333
```

Now the syntax may look kind of weird. We created this altuve variable. And we are storing the battingAverage variable of the function inside of that, because of that. Because it returns an object we can use the object dot type of traversal. So we're able to just call altuve.getCurrentAverage just like this. Just like we are creating a plain user object or some type of object just like we've done before where we create an object call a user and say user.name and it pulls in that value. We can do the exact same thing here when we're returning a function. We're able to call that function just like a normal object attribute which is something very cool. Not a lot of programming languages allow you to do something like this so this is neat. And it worked. So that works out nicely. That worked very well.

Now let's update some things because we have our other function or method. Let's now try to change some values so I can say altuve.update and I'm just going to copy this function or this method because it's a little bit longer one so altuve.updateHitsAndAtBats. I'm going to pass in no hits and 20 atBats. Then we can call this again and that's when the cool thing is this is dynamic. If I leave the code just like it is here we should get 333 for the first console.log, but then we should get a different number for the second console.log, because we increased no hits, and we added 20 atBats. So if I run this you can see it worked.

```
var altuve = battingAverage();
console.log(altuve.getCurrentAverage());
altuve.updateHitsAndAtBats(0, 20);
console.log(altuve.getCurrentAverage());

// 0.3333333333333
// 0.3125
Benefits of Closures

    State Management: Closures can maintain state across multiple
function calls.
    Encapsulation: Closures can hide data and functionality, promoting
modularity and preventing naming conflicts.
```

```
    Dynamic Behavior: Closures can create functions that adapt their
behavior based on their environment.

How Closures Work

When a closure is created, it forms a closed environment that
includes:

    The closure function itself.
    The variables in the surrounding scope (lexical environment) where
the closure was created.

Real-World Applications

Closures are used in various scenarios, such as:

    Callbacks: Functions passed as arguments to other functions.
    Event Handlers: Functions that respond to user interactions.
    Data Privacy: Creating private variables and methods within
objects.

Important:

    Closures are functions that "remember" their environment and
maintain state.
    They are created when a function references variables from its
outer scope.
    Closures are powerful tools for state management, encapsulation,
and dynamic behavior.
```

We have a batting average of 333 and in the second one when we added all those atBats without a hit his average is now.312, So this is working perfectly.

And with updateHitsAndAtBats, it took in its two arguments, it went and grabbed hits and atBats incremented them by whatever we passed in. So if we just passed in a 1 and a 1 it would've increment hits and atBats by 1 and 1. And because we're storing all of this inside of the altuve variable that allows us to do is maintain state. So if you built something like this, say you're building an eCommerce website and you're building a shopping cart you'd be able to do something exactly like this where every time they added a new item to the shopping cart you'd be able to keep track of that and you'd be able to customize their experience based on the values that they are adding into the card.

So this is something that is very powerful and as you start to get into more professional real-world types of programs such as building types of libraries on Node or working with the Express framework or angular or one of those. You're going to see this happening a lot where you are returning not just a few values but you're actually returning objects and then those objects can be called and you can use them however you need to. Whether it's to update values, to add and create values, to perform form submissions, anything like that.

I am excited about this one because when I look at this code this is looking a lot more like real-world application code. This is something I could see where to take out a few of these type values and slide in API calls, and all of a sudden this is communicating with an outside server and you're doing some pretty cool things. So this is exciting because this is something that you could actually be building in not too long and you're going to be building some more advanced behavior. I'd definitely recommend if this is unclear at all, go through it a few times. Customize these functions so you can become familiar with the syntax. Get an idea for how the data flow works and you'll be ready to move on in the course.

```
function battingAverage () {
```

```
  var hits = 100;
  var atBats = 300;

  return {
    getCurrentAverage: function () {
      return (hits/atBats);
    },
    updateHitsAndAtBats: function (hit, atBat) {
      hits += hit;
      atBats += atBat;
    }
  }
}

var altuve = battingAverage();
console.log(altuve.getCurrentAverage());
altuve.updateHitsAndAtBats(0, 20);
console.log(altuve.getCurrentAverage());
```

Coding Exercise

Take the variable roomOne and call the function on it to return the seats remaining.

NOTES:

```
/* Creating methods; an object inside a return value.
One known function configures values, different kind of data which we
need
to manage with another function, a non-defined as the first.
in order to "get" the calculations needed from our desires.

A CLOSURE can help by giving access to previous data which has already
called/returned.
This is what closures do.
Other way to understanding it:
A Closure, a var as a subordinate function
*/


function movieTheater(){
  var seats = 50;
  var seatsSold = 28;

  return{   // THIS UNNAMED FUNCTION IS THE CLOSURE !!!!!""
    remainingSeats: function(){
      return (seats - seatsSold);
    }
  };
}

var roomOne = movieTheater(); // HERE, WE CALL THE FUNCTION BY ADDING
(PUSHING) ITS VALUE TO THE VAR

roomOne,movieTheater(); // HERE IT'S WHY WE CAN CALL THE CLOSURE
FUNCTION WITH A VAR SET OUTSIDE FROM IT.
// HOW?/WHY? We ask for returning the roomOne value parsed through
movieTheater(), and it's this function
// that has a subordinate function (as a closure) which does the logic
needed in this example to calculate
// the remaining seats.
```

3.8 Introduction to HTML Scripting with Built in JavaScript Functions

This guide walks through how to leverage built-in JavaScript functions to alter content on a website.

THEORY:

Understanding the DOM (Document Object Model)

The DOM is a programming interface for HTML documents. It represents the page as a tree-like structure of nodes, allowing you to access and manipulate elements, attributes, and text content using JavaScript.

Example: Changing Text Content.

```
// Access the first element with the class 'b1'
let element = document.getElementsByClassName('b1')[0];

// Change the inner HTML of the element
element.innerHTML = 'Hi there';
```

Explanation

- `document.getElementsByClassName('b1')`: Selects all elements with the class 'b1' and returns a collection (HTMLCollection).
- `[0]`: Accesses the first element in the collection (index 0).
- `.innerHTML`: A property that allows you to get or set the HTML content of an element.

Use Cases for DOM Manipulation

- **Dynamic Content Updates:** Change text, images, or styles on the page based on user interactions or events.
- **Form Handling:** Validate user input, submit forms, and display results without page reloads.
- **Web Page Automation:** Automate tasks like filling forms, clicking buttons, or extracting data.

Example: Dynamic Greeting

```
let userName = prompt("Enter your name:");
let greetingElement = document.getElementById('greeting');
greetingElement.innerHTML = "Hello, " + userName + "!";
```

Important Considerations

- **JavaScript Placement:** Place your JavaScript code in `<script>` tags either within the `<head>` or `<body>` of your HTML document, or in a separate `.js` file.

- **DOMContentLoaded Event:** For scripts that interact with the DOM, it's best to execute them after the page has fully loaded. You can use the `DOMContentLoaded` event for this.

```
document.addEventListener('DOMContentLoaded', function() {
  // Your DOM manipulation code here
});
```

Important:

- JavaScript provides built-in functions to interact with the DOM.
- You can access and modify elements, attributes, and content using JavaScript.
- DOM manipulation is essential for creating dynamic and interactive web pages.

VIDEO:

Since we're talking about Javascript functions I thought it would be fun to take a little bit of a break and talk about some of the built-in functions provided in the javascript language. And since javascript renders inside of the browser many of the functions that it provides also work with your browser itself.

I have a page open on devcamp just where people can ask questions and if I come here and click on inspect this actually gives me the ability to come and check out all of the code on here. Now obviously this is only going to give access to the front end codes such as the `HTML` and `CSS` the server-side code is on the back end and you can see that. However, because javascript does allow us to have all of these cool functions we can actually play around with some things here.

Now some of the things we're going to talk about here make sense more when you get into things like, building angular based apps or react based apps but hopefully can help give you a little bit of foundational knowledge and also help you understand how you can use javascript in the browser. This also comes in very handy when you're wanting to perform tasks such as rendering or dynamically rendering content on the page. And that's something that you'd get into more with things like `Jquery` and frameworks like that but it's still fun to learn what's happening behind the scenes.

The first thing we're going to do is inside of elements here, I can click on body and it's going to show me all of the body inside of it so I can keep on traversing here. And as you notice it highlights different components on the page and it keeps on isolating it down more and more. Now if I click on inspect and actually click on something such as raise your hand right here what this is going to do is show me the exact set of code that I'm looking for. So what I'm looking to do is to write a little script that from the console from the javascript console will allow me to change the text on the page. Now this will not change it permanently. Only something that happens on the server-side can change it permanently. But this is something that you can do on the fly in your own applications and using the console is a great way for testing those kinds of scripts out.

So right here we can see a number of things we can see that this is in an `H2` tag and that has a class of `b1`. And that's something very important because we need to be able to tell our javascript script exactly what to look for. So I'm going to open up the console here and I'm going to start this out. I'm not going to go into great detail on some of the underlying technologies of the framework here. In other words, there is a whole concept of what's called the `document object model` which is pretty extensive and probably could have an entire course written just about that.

And that is how a page is able to be queried and how you're able to dissect certain portions of that page. So some of these things I will give more of a high-level overview on.

The first thing you say is document. Now if I just say that you can see that this gives an object that gives an object of document in this pulls all of the code in. So when we call document what we're telling javascript is I want to have access to all of the code everything that we just saw in elements I want access to that. So that's but we want more than that we want to filter down more than that. So the function that we're going to call, this is a built-in javascript function is called `getElements` we could grab it by ID but this specific element I'm looking for didn't have an ID, did have a class name though. So I'm going to call `getElementsByClassName` and here I want if you remember the class name was a `b1`.

Now if I run this, you can see it pulls in the `h2, b1` which is exactly right here where it says Raise your hand. If I open this up you'll see this is an object. This gives us a number of attributes so it has a link, in other words, if there were multiple classes named `b1` then it would have pulled up all of those classes, and then we could have picked which one that we wanted to work with.

Now inside of this zero with, remember this starts counting at zero. So when it says 0 this essentially just means the only item found, if we would have had multiple `b1` classes all over the page. It would have said zero, one, two, three and it would have isolated that. If I click on that, now we can see all kinds of attributes and it's associated with a key-value pair kind of set up so we could call class name we could call a client top we could check to see if it's drag-able and all kinds of different things. We also have access here, this is important. This is our `innerHTML` this is what we're actually going to be using to change the content. So this is saying this is the text that is in the system and I want the ability in our case to change that. And there's definitely all kinds of other things you can check out and look into. But for right now the most important one is that in `innerHTML` by I definitely recommend for you to play around some of the other ones, so you can kind of see what's available.

Now that we have that, we're going to use a notation called the `bracket notation` to grab that first one. I'm going to say bracket zero which is going to bring me just that very first one, which is the only one on this page. And as you can see that brought it right here. So now we have our class and b1 with Raise your hand.

Now we still have one other thing I want to do which is to actually change the text. So I can say .innerHTML and it all has to be capsulize for HTML.

```
document.getElementsByClassName('b1')[0].innerHTML = 'Hi there';
```

And now I'm going to change this to say hi there and now watch what happens to our h2 tag up here. If I hit return it got changed to hi there. So that's something that is pretty fun. It's very helpful when we're building programs that need to change their content on the fly. Imagine a scenario where you have a welcome screen and you want to put in the user's name or you want to put it in right after they've typed it in. You can have a script that takes the value that they typed in and then using tools like this where you find an ID or a class and then you can have it and set the HTML or set the text value of that to something else just dynamically on the fly.

Another spot where this comes in very handy is building scripts for automating processes. I use that quite a bit, where I will build some type of script. For example, I had to build a script to invite about 700 students to a website called C9 one time. And with that, it would have taken a very long time to invite those users one by one. So instead what I did, is I took all of the users and stored them inside of a variable, and then I created a loop and that loop went over a script kind of like this but instead of just setting an H1 or something.

It filled in the username, the password, it filled in all those things, and it even pressed the button and then it waited for a few seconds for it to load again and then it went and did the same thing over and over and over again for all seven hundred and it made it much faster and much more efficient to invite users instead of doing it manually. So there are all kinds of things that these built-in functions allow.

This is not a course on working with the document object model. We cover those kinds of topics in other courses. This is more on pure programming and Javascript. But I wanted to show you what's possible and show you some of the functions that are available inside javascript because there most likely will be a time when you will want to use those.


```
document.getElementsByClassName('b1')[0].innerHTML = 'Hi there';
```


Coding Exercise

Grab the below paragraph tag by its class name and change the inner HTML of the tag to `Wizards don't jump they float`.

```
<p class="grabThis">The five boxing wizards jump quickly</p>

var weirdSentence = //Write your code here!


/* Never, never, never try to get more than one equality by line !!!

Web API: document

Syntax:
document(.). // creates a new document object on the DOM tree

Full syntax
document.getElementBy...('$the_element')[position].where

Method ‘getElementsby…’:
getElementsByX():
Id
```

```
ClassName
Name
TagName / TagNameNS (given tagName and/or tagName and namespace)


The element must be [selected] even if there's only one element,
starting by zero.

*/
```

3.9 Introduction to the 'this' Keyword in JavaScript

This guide discusses how the 'this' keyword is used in real world JavaScript programs.

In this guide I'm going to introduce and walk through a real world scenario of a topic in javascript development that can be a little bit confusing especially to new developers but also it can be confusing to even more experienced developers just because it has a few caveats and a few things that are more challenging to understand and the key word is `this`. So `this` is a very special word in Javascript. And if you've never seen it before then it might be a little bit hard to understand at first which is why I'm not even going to get into writing code for it right in the middle because I think one of the easiest ways to understand it.

Is actually to be able to see how it's used in a real world scenario. So I'm going to come here I'm on the dev camp library page here and I'm going to come and select one of these elements and one thing you may notice is I have this little hover effect here. And when you hover over one of the library items it gives a short synopsis on what is happening. And I actually use javascript when I was building in this feature. So I want to show exactly how I was able to do that. So I'm going to come here and select one of these items so you can see that we have these images and we also have these buttons. And so we have a bunch of different things that we can select.

Let me open this up in the inspector and just so you can see all of the different items that you have available. You have of course buttons, and then if you open this up you can see that this has that button that you just saw right here. What I want to do is actually because this button is a thing that I'm looking to grab i'm going to select button guide and now switch back to the console i'm going to write a little `Jquery`. Now I know this is not a `Jquery` course but `Jquery` is javascript it's just a layer on top of javascript and so I want to show you kind of a more practical way that you'd be able to use this for a real world app. This is a feature like I mentioned I already just built. So here I'm going to select it so I'm going to look for the button guide and then I'm going to create what's called a click handler which means it's going to look to see and it's going to wait for me to click on this item.

I'm going to say click(function(event). Now do not worry about this. Like I said this isn't a `Jquery` course we're not going to get into all the details on jquery just yet you can go through the jquery course for those kinds of things. But what we have here is essentially we're building a listener that's waiting for an event to happen and in this case it's waiting for us to click on this button guide or on any of the button guides because one thing to note every one of these has this button guide. So when you have a scenario like this where if you scroll down you can see there are dozens of courses. We don't want to hard code in these values so we don't want to say OK, yes this is for the rails course, this is for photo course, this is for an API course.

You wouldn't want to do that because, one that would be very very slow from a development perspective you'd have to go back and hard code each one of those values every single time and it wouldn't be scalable every time that I create a new course. I'd have to go create a new javascript

function to say oh yeah make sure you take in this new course and build this one handler on top of that.

Instead I just add a selector for the button guide that watches for everything and then I wait for a click or in the case of this feature that I built, I waited for a hover event. So inside of it there's a couple things I need to do. First I'm going to take that event and I'm just going to say preventDefault(). That's simply saying I do not want when someone clicks on this, I don't want it to actually go to the next page for the sake of the script. Don't worry about that once again, that's where you're just waiting and listening for something and it's more for the sake of this demo. This is the part I want to show you. So here I want to say console log then do a dollar sign and once again this is jquery. I'll show you the syntax for pure vanilla javascript after this.

I'm going to just console.log(this) the jquery part is the one that has the dollar sign the part to keep in mind is just this. So now if I close this out it's going to look like this all put together.

```
$('.btn-guide').click(function(event) {
    event.preventDefault();
    console.log($(this));
});
```

What this went and did, is it found 17 courses or it found 17 of these buttons that we're looking for this button guide. So it found all of these, and that's kind of helpful but right now it's not what we really need. Imagine if you were tasked with building a feature kind of like this one where you wanted javascript on a hover or on a click or something to make a certain set of descriptions appear. So here we have all these descriptions. How do we know which one was hovered over in order to pull up the content. Well that is where this comes in so handy because what this refers to is the specific item that we're looking to process the specific item that we're looking to do our work with. So imagine in this case what I built in was a hover effect that said, OK, when you hover over something that has this class I want you to do such and such type thing.

In this case I want you to show this set of values but I want you to do it only on this so I only want you to show it when you hover over this rail scores only show the description for this Rail's course if not just keep it hidden. Same thing for each one of these. So coming down to example a little bit more now that we've done all of this. Now watch what happens if I come and I click on one of those buttons which is the one we selected one that preventDefault made it so it doesn't continue to the course.

Now look it printed out this, remember that's what the console log was we said we want you to just print out this object. And so that's what it did. It picked out what, got selected which in this case was, Learn Rails from Scratch course. If I click on here I can see all of the items associated with it. I can come here and see where the base URI is, I can see here that classes that were associated with that button and I can even see where it's pointing to. So everything and you can also get the innerHTML, you could do something like where you hover over and the content changes. Because if you have access to see it here that means you also have access to change it or do anything that you want with it.

So this is pretty cool, this is a way of and I hope this is a helpful example for understanding how this works. And you know let's look at the code just one more time. And so what we have here this is referring to what ever it's looking for. So whatever we click on it's referring to that specific item it is something that gives you the ability to make your code much more dynamic because you don't have to hard code in the values that you're looking for you're actually able to build your selectors in

a way where when someone clicks on something you know what they clicked on. So it makes it so that you don't have to know the future. You don't have to know what content is going to be there.

This really gets at the heart of what makes javascript so powerful is because you're able to allow the user as they're navigating your application to click on, to hover over things to type things in. But because of this you're able to know what they're on and then you're able to build your own custom functionality on top of that. So this was kind of an introduction to this in the next guide we're going to walk through how we can actually program with it and build it into our own programs.

```
$('.btn-guide').click(function(event) {
    event.preventDefault();
    console.log($(this));
});
```

Coding Exercise

Follow the instructions below to utilize the `this` keyword to create a new person with the name of Jordan

```
class Person {
  constructor(name){
    this.name = name;
  }
}

const yourPerson = new Person('enterTheNameInHere');
```

NOTES:

```
/* 'this is used to get the OBJECT value, not the method argument */
```

3.10 How to Use the 'this' Keyword in JavaScript Programs

This tutorial examines how to work with the 'this' keyword in JavaScript programs. As an example, in this guide we'll build an authorization engine by combining object, closures, and the 'this' keyword.

THEORY:

Understanding `this`

The `this` keyword in JavaScript refers to the object it belongs to. However, its value can vary depending on how the function is called. Understanding `this` is crucial for working with object-oriented JavaScript and event handling.

Real-World Example: Interactive Course Descriptions

Imagine a webpage with multiple course descriptions, each with a button to reveal more details. Instead of writing separate JavaScript code for each button, you can use `this` to dynamically target the clicked button and its associated content.

jQuery Example (Illustrative)

```
$('.btn-guide').click(function(event) {
  event.preventDefault();
  console.log($(this));
});
```

Explanation

- `$('.btn-guide')`: Selects all elements with the class `btn-guide` using jQuery.
- `.click(function(event) { ... });`: Attaches a click event handler to each selected element.
- `event.preventDefault();`: Prevents the default behavior of the button (e.g., following a link).
- `console.log($(this));`: Logs the jQuery object representing the specific button that was clicked.

How `this` Works

In the example above, `this` refers to the specific button element that triggered the click event. This allows you to access properties and methods of that particular button, such as its ID, class, or content.

Vanilla JavaScript Equivalent

```
const buttons = document.querySelectorAll('.btn-guide');

buttons.forEach(button => {
  button.addEventListener('click', function(event) {
    event.preventDefault();
    console.log(this); // 'this' refers to the clicked button
  });
});
```

Benefits of Using `this`

- **Dynamic Targeting:** `this` allows you to target specific elements without hardcoding their IDs or relying on their position in the DOM.
- **Code Reusability:** You can write a single event handler that works for multiple elements.
- **Maintainability:** Makes code easier to maintain and update, as you don't need to modify the JavaScript every time you add or remove elements.

Important notes:

- The `this` keyword refers to the object it belongs to.
- In event handlers, `this` typically refers to the element that triggered the event.
- `this` is essential for writing dynamic and reusable JavaScript code.

VIDEO:

Now that you've seen in a real-world scenario how you can use the word this. Let's get into how we can actually use it in pure vanilla javascript and we're going to talk about one of the most common ways to use it which is inside of objects. So let's build out a program and we're going to call the object a guide. This is going to have key-value pairs but we're also going to include our own methods inside of the object so the goal of this guide is going to be able to have a title and content.

It's supposed to have data but then it also is going to have an authorization engine. So what it's going to do is check to see if the viewing user is a paid user or if they're a free user and if they're a paid user they'll render the content out if they're a free user, they won't. They simply will only show

the title, not the content. So that's something that's pretty common to build in real-world applications so here I'm going to give it a title called The Guide to programming and then give it content and we can just say content will go here. Now let's build our methods, the first one is going to be one that I'm going to call visibleToUser. And this is going to be as all of the other types of methods are anonymous functions that are assigned to whatever we name them.

In this case, we named it visibleToUser and this is going to take in one argument which is going to be viewingUserRole and inside of this, we're going to have a conditional. I'm going to say if viewingUserRole which is our argument is triple equals two paid then we're going to return true and else we're going to return false. And that is our visibleToUser method.

```
var guide = {
  title: 'Guide to Programming',
  content: 'Content goes here...',
  visibleToUser: function (viewingUserRole) {
    if (viewingUserRole === 'paid') {
      return true;
    } else {
      return false;
    }
  },
}
```

Now we're going to build a method to render the content out. So we call this renderContent. This is going to take in a user role and then inside of this we're going to build out our system. Now let me just clear up the indentation. And now we're ready to keep on building the method. So renderContent inside of this function. I'm going to check to see and say if and then we could say you may think that we'd be able to say something like, (visibleToUser(userRole). Now, what this seems like it would do is call our method up above and that is logical especially if you've come from another programming language. However, there is a little bit of a catch here and this is where this comes in. If we just tried to run this exactly as we have right now javascript is just going to skip over this. They are going to ignore what we're saying here because we need to be more explicit.

We need to say this.visibleToUser because it needs to know the exact instance that we're referring to. If we just say this broad generic visibleToUser function it's going to look through the global namespace and check to see if there is a visibleToUser function.

If not it's just going to return undefined we're not going to have anything here. And this isn't going to work. But when we say this it knows we're referencing this objects visible to use or method that is the key right there. We are referencing this specific instance of this object because this, in a traditional kind of application what usually is going to happen is you're going to have hundreds or thousands of these guide instances. If you just called a generic form of visibleToUser. It's not going to know which one you're referencing, we need to tell it that we're referencing this one for this instance. Now that we have visibleToUser, if that is true now inside of it we want to perform some logic say console.log and once again we're going to use `this` and this is a reason why I picked out this example because I want to show how you can use `this` both for methods and also for data.

I can say this.title, and then here and just put a dash with some spaces in between followed by this dot content. And that is all you need to do there. That is for if the user is paid, in other words, if that is the visibleToUser comes back as false, I want to say this.content and set it equal to an empty string and then I'm just going to say console.log this.title. And so this one is going to be the exact same and the reason why I'm doing this obviously since we're doing in other console log statements it is very easy to just get rid of this entirely. But I wanted to do this so you can see a few different things.

One we've talked about how you can call methods and also how you can call attributes but you also have the ability to change the values of attributes so that's also something important I wanted to show you all three of those different ways that this can work.

With all of that in place, I think we're ready to test this out. So let's say user equals role and I'm just going create a basic object. And for the first one let's say paid and then I'm going to call a guide and then render content. And remember we just have to pass in the user role. And this should print now the full guide to programming and content.

```
      console.log(this.title + " - " + this.content);
    } else {
      this.content = '';
      console.log(this.title + " - " + this.content);
    }
  }
}

user = { role: 'paid' };
guide.renderContent(user.role);

// "Guide to programming - Content will go here"
```

Let's test this out to see if it works, and look at that! It worked.

Now, what happens if it's a free user? If change the user role to free and I hit run again, it shows "Guide to programming - " and the rest is blank. So this is working perfectly. We have our own little authorization engine here where it can check to see and based on a user's role it can dynamically generate a different type of content it can render content in much more of a dynamic fashion because we have the ability to call these other methods.

This also comes in so handy when it comes to being able to encapsulate and organize our code properly because we don't have to worry about trying to pass all of that logic inside of here. Imagine if we had to put all of this code here and then input that inside of the rendered content that would really not be a great way to organize our code. It could start to get very messy quickly but by being able to leverage the concepts like `this` where you have the ability to say I want to create methods that specifically speak to this object. They completely ignore other objects in the global namespace. They're only talking about this one object that gives us the ability to be very explicit with the types of methods we're calling to know and be confident about the type of data attributes that we're working with. This definitely falls in the realm of the way that javascript programs are being built in our modern times.

```
var guide = {
  title: 'Guide to Programming',
  content: 'Content goes here...',
  visibleToUser: function (viewingUserRole) {
    if (viewingUserRole === 'paid') {
      return true;
    } else {
      return false;
    }
  },
  renderContent: function(userRole) {
    if (this.visibleToUser(userRole)) {
      console.log(this.title + " - " + this.content);
    } else {
      this.content = '';
      console.log(this.title + " - " + this.content);
    }
  }
```

```
}

user = { role: 'paid' };
guide.renderContent(user.role);
```

Coding Exercise

Use 'this' to run the code and determine how many seats are left.

```
var seats = {
  seats: 50,
  seatsSold: 28,
  remainingSeats: function(){
    return (this.seats - this.seatsSold)
    },
  enoughSeats: function(){
    if(this.remainingSeats() > 0){
      return // use this and seats to return the number of seats left.
    }
  }
}
seats.enoughSeats()
```

NOTES:

```
var seats = {
  seats: 50,
  seatsSold: 28,
  remainingSeats: function(){
    return (this.seats - this.seatsSold); // this.seats = the value
from seats, NOT THE OBJECT SEATS
    },
  enoughSeats: function(){
    if(this.remainingSeats() > 0){
      return this.remainingSeats(); // this.remainingSeats(), the
VALUE RETURNED FROM THE CLOSURE, not the closure
    }
  }
};


seats.enoughSeats();
```

MODULE 4.

ARRAYS and DATA STRUCTURES

4.1 Section Introduction: Introduction to JavaScript Arrays

Now, if you've never heard of an `array` or you've never used them before at a high level: what an `array` is, is it is a collection. So it gives you the ability to store a collection of data. So whereas with a `standard variable`, so say when you store a `string` like a name that is just a single data object it's a single string.

Well, what happens if you need to store a large number of names. Well, that is where arrays come in. As you go along your coding journey, you're going to discover that arrays are used throughout the entire JavaScript world.

Any time that JavaScript is going to store a list of items, it's typically going to be in an array or a data structure very similar to it. For example, if you're building out a directory site kind of like a Yellow Pages or a Yelp kind of site, and you contact an `API`. That API is going to send you a list or a collection of items back.

The way that JavaScript is going to interpret that is it's going to place that inside of an array. So if you're calling a database, if you're calling an API, if you're getting any kind of collection, JavaScript is just about always going to put it into an array. That's why it's so important to be able to understand what arrays are and also how to work with them.

The real-world analogy that I like to give to really help explain and solidify why arrays are so important is: imagine that you're going to the grocery store. If you go to the grocery store for just a few items, then you can just pick those up put them in your hands, and then carry them to the cash register.

If you go and you have 100 items to get or even 12 items to get, any number that you can't really carry in your hands, you need something to put those into. You need a shopping cart typically.

Well, inside of JavaScript that's what you use `arrays` for. You can really only store so many items inside of a `variable`, such as just you can put one number in a variable, or you can put one string in a variable. You can only really have so many variables before it starts to get a little messy.

If you have the same data type, so if you have a bunch of usernames or restaurant listings or blog posts, then typically you're going to store those in an array. In the same way that when you go to the grocery store, it starts to get very unwieldy if you started carrying all of your groceries just in your hands.

Same thing holds true for JavaScript. You want to be able to have a data structure that you can wrap all of those objects in, then you can store that single array inside of a variable, and then you can work with it. With all that being said, let's dive into the code and start building out some arrays in JavaScript.

4.2 Creating Arrays

This introductory lesson walks through the various ways that you can create arrays in JavaScript. Additionally, we examine how we can query elements in arrays using the bracket syntax.

THEORY:

What are Arrays?

Arrays are ordered collections of data. They can hold elements of various data types, such as numbers, strings, objects, and even other arrays.

Creating Arrays

1. **Array Constructor:**

   ```
   let myArray = new Array(3); // Creates an array with 3 empty slots
   ```

2. **Array Literal Notation (Recommended):**

   ```
   let myArray = [1, 2, 3]; // Creates an array with three numbers
   ```

Accessing Array Elements

You can access array elements using bracket notation and the element's index. Remember that arrays are zero-indexed, meaning the first element is at index 0.

```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]); // Output: "apple"
console.log(fruits[2]); // Output: "cherry"
```

Storing Array Elements in Variables

```
let firstFruit = fruits[0];
console.log(firstFruit); // Output: "apple"
```

Mixed Data Types

JavaScript arrays can hold elements of different data types.

```
let mixedArray = ["hello", 10, true, { name: "John" }, [1, 2, 3]];
```

Nested Arrays

Arrays can be nested within other arrays.

```
let nestedArray = [[1, 2], [3, 4], [5, 6]];
console.log(nestedArray[1][0]); // Output: 3 (accessing the first
element of the second inner array)
```

Accessing Elements in Mixed Arrays

```
console.log(mixedArray[3].name); // Output: "John" (accessing the name
property of the object)
console.log(mixedArray[4][1]); // Output: 2 (accessing the second
element of the inner array)
```

Functions in Arrays

You can even store functions in arrays and execute them.

```
let myFunctions = [
  function greet() { console.log("Hello!"); },
  function sayBye() { console.log("Goodbye!"); }
];

myFunctions[0](); // Output: "Hello!" (calling the first function)
```

Why Use Arrays?

- **Data Organization:** Arrays provide an organized way to store collections of data.
- **Iteration:** Arrays are easily iterable, allowing you to loop through their elements.
- **Flexibility:** JavaScript arrays can hold elements of different data types, providing flexibility.

Key Takeaways

- Arrays are ordered collections of data.
- Create arrays using array literal notation (`[]`).
- Access array elements using bracket notation and zero-based indexes.
- JavaScript arrays can hold elements of mixed data types, including nested arrays and functions.

VIDEO DIALOG:

In this guide, we're going to start working through what arrays are in JavaScript and how we can work with them.

There are two main ways that you can create an array and we're going to use the first syntax which is essentially a generated syntax where we create a new array object. And so I'm going to store it in a variable called `generatedArray` and the syntax for this is `new` which is a special keyword and then array. Now there are a couple of ways to do this. Say that you want an array with three elements inside of it, we can pass it in just like this.

```
var generatedArray = new Array(3);
```

And this will create the array for us.

If I say `generatedArray` now you can see that we have 3 items. They're all undefined. But we do have a collection. We could even call generatedArray length and you can see that we have three items in this array. Now having 3 undefine items may not seem very helpful. So let's go and let's create a new array with some names so I can create some baseball player names here.

And now if I run

```
var generatedArray = new Array('Altuve', 'Correa', 'Spring');
```

this generatedArray you can see that we have a collection of three individuals and three strings. So that is one way that you can generate an array. I don't use that way too often and the only time I'll usually use it is when I want to create an array and I don't know what the values are going to be, but I happen to know how many elements are going to be inside and then I use exactly what I did up here.

The more common way that I create an array is by creating what is called the `array literal syntax`. And so here is a VAR literal array. Obviously you could call it anything you want and here it is simply going to be using the brackets. And so I can put inside of these three items and run it. And now I have a literal array just like that.

```
var literalArray = [1, 2, 3];

literalArray; // (3) [1, 2, 3]
```

Now I could obviously mix these. So those are using integers but if I wanted to I could put strings inside of it. So if I go literal array again I can reassign it to these values. And now if I call literal array it has Altuve, Correa, and Spring which is supposed to be Springer the baseball player.

```
var literalArray = ['Altuve', 'Correa', 'Spring'];

literalArray;  // (3) ['Altuve', 'Correa', 'Spring']
```

I can also do just like before, I can call link that has that attribute associated with it so that those are the two most common ways.

Now I've shown you how you can have integers and I've shown you how you can have strings. But JavaScript is incredibly flexible and you can mix and match the elements in the data types inside of your arrays. So let's create a new one called mixed array. And with this mixed array you'll see we can combine anything that we want. For the most part. So I can say hi and put a integer. I can't even put another array. So I could but the ABC's is right here and this is going to be a nested array inside of it.

Make sure you separate each one with the comma and then from there I can even put in objects. Here I can put an object with a name, close the object out and the last one we're going to do may seem a little bit tricky but I can actually put a function in here as well. I could say function greeting and then inside of it console log 'hey there' and then also close it off obviously.

```
var mixedArray = ['Hi', 1, ['a', 'b', 'c'], { name: 'Kristine' },
function greeting() { console.log('hey there');}]
```

And we have our mixedArray, so that should show you could put literally anything you want. It's a collection of data but JavaScript is a much more flexible with what you can put inside of your arrays than many other languages. Many languages like C or Java in those ones you have to declare the type of elements that are going to be inside of an array and that's because they have very strict compiler requirements. whereas, the JavaScript engine is much more flexible and what it allows. So that's how you can create arrays.

Now let's talk about how we can actually get things out of the arrays. So let's start with our most basic one. If we go back to literal array here the syntax for getting items out is what's called the bracket syntax so you put two brackets right here. And then inside of it you call the index of the item that you want to pull out. Early on in this course I talked about indexes in computer science starting with 0 instead of 1. We discussed that when we're talking about how we could count the characters and grab characters from a string.

We can follow the same pattern when working with arrays so right here if we want Altuve, we can pass in zero and remember it's because arrays and indexes in general start with zero.

```
literalArray[0]; // "Altuve"
```

If I hit return, that pulls out Altuve and now I can use it however I want. So a very common pattern you'll see is to do something like this. I can say player name and then call the literalArray and this time let's go with the next item which is going to be 1. So I store it and now if I call player name you can see it's stored with Correa and that is very common kind of pattern that you will see in development.

```
var playerName = literalArray[1];

playerName; // "Correa"
```

Now that is helpful but you may still kind of be wondering when in the world am I going to use this? That's a very common question to ask especially if you are new to development. Arrays are incredibly handy for a number of scenarios. One of the most common that I use them for is with database queries.

Usually when you're building an application and you make a query to a database or to an API when you receive that data back there has to be a standard representation for how that data is sent back to you. One of the most common is to have data sent as an array and you can then loop over that data show it on the screen. And so that's one of the reasons why it's so important to understand the way that arrays and collections work.

Now let's move down the list a little bit. So we talked about how we can grab a single element. Let's also talk about how we can work with some of those more complex ones.

Remember we have our mixedArray and if I open this up you can see all of the different things that we have inside it. In the zero index, we have "Hi" in the first index, we have the integer 1 then we have an array that has an array inside of it. Then we have a object with a key value pair of name and then Kristine and then we have a function called greeting.

So how exactly can we call some of these other items. Well we already talked about how we could call the first two. Those are pretty basic where we in a call just mixed array and do something like that with the bracket syntax with the next one it may seem like this would be more tricky when you have a nested array. So what we're going to do is pass in the index 2 and whenever you want to

reference that that's one thing I love about the javascript console is it gives you a really nice reference point for your indexes and different things like that.

So here I want the array but let's say that I want the "C" in the array and I know it's at the second index, I can pass in a second set of brackets so whenever you have something nested like this then you can do your very first call your very first query and then use the bracket syntax again and chain these together. Now run this and it returns "C" and you could store that in a variable or whatever you want to do with it. When you're working with nested arrays.

```
mixedArray[2][2]; // "c"
```

Now let's talk about our object. We know it's in the index of 3, and because of that we're going to receive just a plain object back. So if I just say 3, this gives us an object back so don't let the end of the bracket syntax intimidate you because all you need to do is treat it like we've been treating objects this entire course. You can use the dot syntax and chain it together. Now you can have access to that name.

```
mixedArray[3]; // Object {name: "Kristine"}

mixedArray[3].name; // "Kristine"
```

Now let's talk about the last one because this one I've seen trip people up a little bit. So we have our greeting and if you just call 4 that is only going to return the function greeting. But what if we actually want to call the function which is a more standard thing. Well it's going to be the exact same way that we had call a regular function. We've queried it and now we just put our nice parends at the end and it prints out 'hey there' and the function is executed.

```
mixedArray[4]; // function greeting( ) { console.log('hey there');}

mixedArray[4](); // hey there
```

Now go a little bit further if you want to do things like be able to have methods inside of objects that is a very common thing to do and you can have those include it and arrays. So one of the easiest ways of understanding the way arrays work in JavaScript is just think of them as a collection of all the regular items that you've been using this entire course. So there's nothing more special about them than that. They collect everything you can put things inside of it.

It's just a way of storing it, when you want to store multiple things inside of the same variable you can. But beyond that you can treat them exactly the same way as when they were just kind of one off items stored inside of variables. So the main thing to understand is how you can properly query each one of those items just like we walk through right here.

Coding Exercise

Create an array that has 3 elements, there must be a least one number within the array.

```
var myArray = []
```

NOTES:

```
var myArray = [1, 'Potato', 'Fried']; // mixed array


/*
var myArray = [
    { 32 },
    { Math.PI },
    { 'string' }
];

The problem here is that {closing each array element} creates a new
object
*/
```

4.3 Adding and Removing Array Elements

This guide walks through how to use the push, pop, shift, and unshift functions in JavaScript in order to add and remove array elements.

THEORY:

Adding Elements

- **`push()`:** Adds one or more elements to the **end** of an array and returns the new length of the array.

```
let fruits = ["apple", "banana"];
fruits.push("cherry");
console.log(fruits); // Output: ["apple", "banana", "cherry"]
```

- **`unshift()`:** Adds one or more elements to the **beginning** of an array and returns the new length of the array.

```
fruits.unshift("mango");
console.log(fruits); // Output: ["mango", "apple", "banana",
"cherry"]
```

Removing Elements

- **`pop()`:** Removes the **last** element from an array and returns it.

```
let lastFruit = fruits.pop();
console.log(lastFruit); // Output: "cherry"
console.log(fruits); // Output: ["mango", "apple", "banana"]
```

- **`shift()`:** Removes the **first** element from an array and returns it.

```
let firstFruit = fruits.shift();
console.log(firstFruit); // Output: "mango"
console.log(fruits); // Output: ["apple", "banana"]
```

Why Use These Methods?

- **Efficient Modification:** These methods provide efficient ways to modify arrays without manual index management.
- **Stack and Queue Implementation:** `push` and `pop` can be used to implement stack (LIFO) behavior, while `shift` and `unshift` can implement queue (FIFO) behavior.
- **Dynamic Lists:** These methods are essential for managing dynamic lists, such as to-do lists or shopping carts.

Example: Managing a To-Do List

```
let tasks = ["buy groceries", "pay bills"];

// Add a new task
tasks.push("schedule appointment");

// Complete the first task
let completedTask = tasks.shift();
console.log("Completed task:", completedTask);

console.log(tasks); // Output: ["pay bills", "schedule appointment"]
```

VIDEO:

So now that you know how to create arrays, let's talk about how we can actually work with them. So I'm going to create a pretty basic array it's going to be the same one I talked about with the baseball players, so you can have Altuve, Bregman, Correa, and Springer.

```
var arr = ['Altuve', 'Bregman', 'Correa', 'Springer'];
```

So now we have our array I can call array length and as you can see we are at four.

```
arr.length; // 4
```

Now to reiterate whenever you see an array it has just like a string a built-in property called Link. Now, this is something that can kind of trip up new developers sometimes and that is that they think the length is a function because in many other languages length is a function. And so you'll see many times where someone tries to do array length and that's going to throw an error because it's not a function it is a property of the array.

That's a reason why when we click down here and we see it we see length is one of the built-in items we also have a lot of other things. This shows all of the various functions that we can call on arrays, this is very helpful and this is some of these are what we're going to go through in this guide. Specifically the ones we're going to hit our push and pop. So these two right here and then we're going to use shift and unshift. So we have shift right here and unshift right here.

I definitely recommend for you to explore each of these methods, we'll get into some of the more especially when we get into our loops and iterations section because some of these that's what they do. Such as "forEach" and "filter" and some of those. et's talk about what pop does, so if I go array.pop and this is a function so I'm going to call it just like this, this is going to return the very last item from the array.

```
arr.pop(); // "Springer"
```

So this is very helpful for a couple of reasons. So now if I do array you can see we now have three items and Springer is no longer there.

```
arr; // (3) ['Altuve', 'Bregman', 'Correa']
```

This is helpful because there are many times where you want to iterate over an array and you want to clean it out. Imagine building a To-Do List application or something and as you are going through you may want to pop them off that's the reference of the name we're popping items off of this array stack right here. The other nice thing is because you notice and this is the reason why

we're using the javascript console to see this is when I do array.pop it pops Springer off but he also returns them.

So let's see how we can add something back in then pop them off again. So just like we have pop, we also have a function called push. So say that I wanted to put someone else in, so I wanted to put Bagwell in and add him to the list.

```
arr.push('Bagwell'); // 4
```

You can see this returns 4 which means that our array now has 4 elements and now we have Bagwell here at the end. If I want to store that variable so if I want to call pop, I want to remove that item but I also want to know what I removed and maybe store in a variable then I can do that.

```
var elementPopped = arr.pop(); // undefined

elementPopped; // "Bagwell"
```

Now I have element pop is going to equal what I popped off of that array. So this is something that is very common whenever you're iterating over a collection and you're wanting to remove items off using pop and push are pretty important.

Now I told you we're going to talk about four functions in this guide. So far we've talked about two and the other two shift and unshift are pretty much the contrapositives of push and pop, meaning that push and pop give you the ability to add items or remove items at the end of an array. Shift and unshift do the exact opposite, they allow you to remove items and add items at the front of the array.

So if the array content order is important to you then you can use these other items so say that I want to pull Altuve off the list. I can say

```
arr.shift(); // "Altuve"
```

this is going to return Altuve the same way that when I pop Bagwell and Springer off it stored it in a variable and returned it. Same thing here we were able to pull it off, now if I call

```
arr // (2) ["Bregman", "Correa"]
```

If I want to add an item to the list I can

```
arr.unshift('Kyle'); 3

arr; // ['Kyle', 'Bregman', 'Correa']
```

Now there are three elements and you can see Kyle is now at the beginning. So this is a very helpful way of being able to work with arrays where we can remove items we can add them in and we can put them at the beginning, or the end of the collection.

```
var arr = ['Altuve', 'Bregman', 'Correa', 'Springer'];

arr.pop(); // "Springer"

arr; // ['Altuve', 'Bregman', 'Correa']

arr.push('Bagwell'); // 4

arr; // ['Altuve', 'Bregman', 'Correa', 'Bagwell']

arr.shift(); // ['Bregman', 'Correa', 'Bagwell']

arr.unshift('Kyle'); // 4
```

```
arr; // ['Kyle', 'Bregman', 'Correa', 'Bagwell']
```

Coding Exercise

Create an array that has 4 elements and then push in a fifth element to the end of the array that says "Bagels".

```
let array = //Write your code here
```

NOTES:

```
/*
SPLICE method uses:
1. DELETE   \
2. ADD       => an element from an array.
3. REPLACE  /

SPLICE method syntax:
varName.method(initialIndex, howManyIndexToSplice, value1, value2,
value3, value4, ... )
*/
```

4.4 Splice, Push, Pop, Shift, Unshift Functions to manage Specific Array Elements

This guide teaches you how to implement the splice, push, pop, shift, unshift functions in JavaScript in order to remove items at specific locations in an array. These methods are essential for managing dynamic arrays and implementing data structures like stacks and queues.

THEORY:

Adding Elements

- `push()`: Adds one or more elements to the end of an array.
- `unshift()`: Adds one or more elements to the **beginning** of an array.

Removing Elements

- `pop()`: Removes the last element from an array and returns it.

```
let lastFruit = fruits.pop(); console.log(lastFruit); // Output: "cherry"
console.log(fruits); // Output: ["mango", "apple", "banana"]
```

- `shift()`: Removes the **first** element from an array and returns it.

```
let firstFruit = fruits.shift();
console.log(firstFruit); // Output: "mango"
console.log(fruits); // Output: ["apple", "banana"]
```

Why Use These Methods?

- **Efficient Modification:** These methods provide efficient ways to modify arrays without manual index management.
- **Stack and Queue Implementation:** `push` and `pop` can be used to implement stack (LIFO) behavior, while `shift` and `unshift` can implement queue (FIFO) behavior.
- **Dynamic Lists:** These methods are essential for managing dynamic lists, such as to-do lists or shopping carts.

Example: Managing a To-Do List

```
let tasks = ["buy groceries", "pay bills"];

// Add a new task
tasks.push("schedule appointment");

// Complete the first task
let completedTask = tasks.shift();
console.log("Completed task:", completedTask);

console.log(tasks); // Output: ["pay bills", "schedule appointment"]
```

VIDEO DIALOGUE:

So far we've covered how to create arrays and then also how to add in remove from the front or the back of the arrays. But we haven't talked about how we can actually remove items from the middle of them which is something that you have after do on a relatively regular basis.

```
var arr = ['Altuve', 'Bregman', 'Correa', 'Springer'];
```

I'm going to paste in the array right here, It's one we've been working with our array of baseball player names. And here I want to pull out and then remove Correa. So right here. Remember indexes start at zero. So we have an index of 0 1 2 and 3. The way that we can do this is by using what is called the splice function.

I could do something like this where I say array splice and then it takes two arguments. The first is the index and the next argument is how many items we want to remove. So in that case I would say two and then one if I just wanted to remove Correa. Then we'd be left with an array of three items.

```
arr.splice(2, 1);
```

However, that may not seem like the most intuitive thing because in most cases you don't know the exact index so you have to find it. So you don't know that Correa is 2, imagine this is coming in from a database query and this has to be dynamic. You can't hard code 2 in but you can search for one of those elements and that's what we're going to start off with.

So I'm going to say var foundElement and then this is going to be set equal to arr and then I'm going to use a function called indexOf. So what indexOf does is it allows you to search by value. So I can pass in Correa right here and this is going to do is it's going to return the index that it finds Correa at.

```
var foundElement = arr.indexOf('Correa');

foundElement; // 2
```

So if I hit return now my found element is equal to 2 because that is the index Correa is out. So now what I can do is say arr.splice and then just put foundElement and then if I only want to remove Correa I can just do 1.

```
arr.splice(foundElement, 1); // ["Correa"]
```

So if I hit return now it pulls Correa out and you can see that it pulls it out. This is a kind of important note splice always returns an array. So even though we only called for one item we and

we only got one item it got returned to us as an array. Technically it returned an array of length 1 and that means that we have to be careful.

Imagine a scenario where I wanted to do arr.splice and I pass in the index and then I pass 1 in and then I want to do something like calling a function on that. I expect it to be a string but it's not, it returns an array. I'd have to do something like that and then I'd have to call with the bracket syntax zero to grab that element. Just a little caveat that can be tricky, I've run into scenarios where I thought I was getting a string but ended up getting an array of one element, and then it wouldn't accept the functions I was sending to it. Until I found out that OK I am not working with a string I am working with an array.

So this is just a little thing that I have experienced in real-world applications. So that is how we can run those in now we can also do other things so say I have an array and say I want to take out the remaining items. I can do arr.splice and this is going to start at index 1 and I want to take out 2 items.

```
arr; // ["Altuve", "Bregman", "Springer"]

arr.splice(1, 2); // ["Bregman", "Springer"]

arr; // ["Altuve"]
```

This is going to return an array of 2 and now if I call arr again it only contains Altuve. So we now have removed each of the items except the very first 1 but instead of using functions like pop or shift to remove those items, we were able to select items much more specifically based on their value like we did right here and also their index. Lastly, we have the ability to not just pull one item off but to pull off as many items as we want.

```
var arr = ['Altuve', 'Bregman', 'Correa', 'Springer'];
var foundElement = arr.indexOf('Correa');
foundElement; // 2
arr.splice(foundElement, 1); // ["Correa"]
arr; // ["Altuve", "Bregman", "Springer"]
arr.splice(1, 2); // ["Bregman", "Springer"]
arr; // ["Altuve"]
```

Coding Exercise

Use the method splice on the array to leave the first 3 values in the array, and have the splice return "Springer".

```
var array = ['Altuve', 'Bregman', 'Correa', 'Springer'];
```

NOTES:

```
/*
SPLICE method uses:
1. DELETE   \
2. ADD       => an element from an array.
3. REPLACE  /

SPLICE method syntax:
varName.method(initialIndex, howManyIndexToSplice, value1, value2,
value3, value4, ... )
*/
```

MODULE 5.

LOOPS and ITERATORS

5.1 Section Introduction: Introduction to JavaScript Loops

In this section of the course, we're going to dive into Javascript loops.

Now, a `loop` in programming is a process where you take a collection of data and then you iteratively go through each one of those `elements`.

Let's imagine a real-world scenario where you're building out some kind of application. You've called a database, that database has sent you back the set of records, so it could be usernames or anything that you're calling the database for. That is going to be stored in an `array`.

Now in order to render those on the page, so if you're building out a web or mobile application, you need to be able to `loop` through that `collection`. So you would take your collection, and then you would use one of the `looping mechanisms` that we're going to talk about.

I'm going to walk through about four or five different options, and I'll talk about when you'd want to use one over the other. At the end of the day, they pretty much all are going to perform the same general purpose; which is to iterate over that collection, and then be able to show that data.

If you're building out a web application, you would iterate over a collection of data, and then you would show those names or that data onto the screen.

That's going to be a very common process, and part of the reason why I saved this looping section to go right after the array and data structures section is specifically because: **if you're working with loops, you're almost always going to be doing that in conjunction with an array**.

Now that you have a good high-level view on what loops are and when you'd want to use them, let's get into the code and start building them out.

5.2 ‘For’ Loops

This lesson walks through the three types of For loops provided by JavaScript: traditional for loops, the for in loop, and the functional forEach loop.

THEORY:

Traditional `for` Loop

The traditional `for` loop is used for iterating over arrays or performing a task a specific number of times.

```
let players = ['Altuve', 'Bregman', 'Correa', 'Springer'];

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

Explanation

- `let i = 0;`: Initializes a counter variable (`i`) to 0.
- `i < players.length;`: Sets the loop condition. The loop continues as long as `i` is less than the length of the `players` array.
- `i++`: Increments the counter variable after each iteration.
- `console.log(players[i]);`: Accesses and prints each player using the counter variable as the index.

`for...in` Loop

The `for...in` loop is specifically designed for iterating over the properties of an object.

```
let player = { name: 'Altuve', position: '2B', number: 2 };

for (let key in player) {
  console.log(key + ": " + player[key]);
}
```

Explanation

- `let key in player`: In each iteration, the `key` variable is assigned the name of a property in the `player` object.
- `console.log(key + ": " + player[key]);`: Prints the property name and its value.

`forEach` Loop

The `forEach` loop is a higher-order function available on arrays. It provides a more concise way to iterate over array elements.

```
players.forEach(function(player) {
  console.log(player);
});
```

Explanation

- `players.forEach(...)`: Calls the `forEach` method on the `players` array.
- `function(player) { ... }`: A callback function is passed to `forEach`. This function is executed for each element in the array.
- `player`: The `player` parameter represents the current element being processed in each iteration.

Choosing the Right Loop

- **Traditional `for` loop:** Offers more control over iteration and is suitable for scenarios where you need fine-grained control over the counter or need to iterate a specific number of times.
- `for...in` **loop:** Best suited for iterating over object properties.
- `forEach` **loop:** Provides a concise and readable way to iterate over array elements, especially when you don't need explicit access to the index.

VIDEO:

One of the most common ways to use loops in javascript development is when you are looping through collections of data. That's where our first example is going to take us. I created a player's array right here and it's just an array of strings.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];
```

We're going to loop over it and I'm going to show you two different ways that you can do this. We're going to use two different types of what are called "for loops."

The syntax is going to be for and then inside of this, we have to put an entire expression. We have to declare three things, first, we have to declare a variable that is going to be used throughout the loop, then we have to declare a condition which means we want you to keep looping until this condition is not met and then the last one is some type of incrementor.

If you've never heard of any of those terms at all don't worry we're going to go through this and we'll go through this one pretty slowly. First thing like I said we need to declare and assign a variable. I'm going to say var i = 0.

```
for (var i = 0; )
```

The next thing we're going to do is we have to declare a conditional. Now what this condition is going to do is as it's looping with each iteration it's going to check to see if this condition is true. if it's true it's going to keep going, as soon as the condition is false, then it's going to exit out of the loop. So I'm going to now say i is less than players.length.

```
for (var i = 0; i < players.length;
```

So what I'm saying here is we have our players array you know that we can call .length on this to give us the full list of items so here it's going players length is going to be 4. So what we're saying is I want you to loop through this entire array until I which starts at 0 is greater than or equal to players length. So as long as i is less than players length then it is going to keep going.

So how in the world does i change? Well, that is the last part of the expression here. So now I'm going to say i ++, remember our ++ incrementor this is where it comes in very handy. With each iteration with each time we go through the loop, I start to zero but then the next time it goes through the loop it is going to change to 1. Then it's going to check to see has this changed, is this still true? If so, we're going to keep going as soon as in this case as soon as it gets to the spot where i is equal to or greater and then players length which is in this case 4, it's going to exit out.

One little caveat if you've never worked with loops before this may seem kind of confusing because we know players length is going to be 4 but we're saying that i is less than players length then that's what we're looking for. Why wouldn't it be less than or equal to? we're going to get into an example on why that is.

The next thing you do is you put in curly brackets and we'll say console log players and remember how we can grab the element is by being index and in this case, i, because it's starting at 0, represents the index.

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

If I hit run this is going to run through and it worked we have Altuve, Bregman, Correa, and Springer.

So how exactly is this working because a logical thought might be that players length because it's 4 and we have 4 items in here that we would want this to be less than or equal to. Well if I hit clear and run this again you'll see that it worked. We have Altuve all the way through Springer but the last one it returns undefined. The reason for that is because remember our indexes start at zero.

So whatever your length is in your array it's always going to be one greater than the last index value. So this starts at 0 that is assigned when it's going through and it's hitting Altuve. When it comes back up, remember it gets changed and it's going to get incremented to 1 that represents the

index for Bregman. Then it goes through again, it's going to be 2 which is Correa then Springer is going to be 3. So in the case where this is put together properly, it is still going to go through.

That is our last item, as soon as it comes back up and it gets changed to 4 it checks and says OK is i, in this case, is 4 less than the player's length. No, it's not. OK. Exit the loop we're done and there's nothing else that's going to happen. So that's exactly what's happening. That's what we want, this is something I'm harping on this a little bit because it's very important. As you look through other people's code you're going to see so many times where they call something like array length minus 1.

And if you haven't seen this and you don't understand why that's necessary you're going to wonder why they have i minus 1 all over the place. Technically you could do something like this, you could say this would work if I said i was less than or equal to the players length minus 1. If I run this and now it all works properly so we could either say i is less than or equal to the player's length minus 1 or the way that I usually do it is I just say that i is less than the players length. This is to me it's a little bit easier to read.

So this is the first way of doing it and it's OK. For loops can be very handy, they've been in computer science for decades and decades. However, I personally whenever I'm using this type of programming construct where I'm looping over a collection I like to use a little bit more modern kind of for loop, and that is called the for-in loop.

So I'm going to get rid of this I'll keep it in the show notes so you can access it there.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];
```

And so how for in loops work is you say for then you declare an iterator variable. The common convention is to use whatever your array has a plural name like players the common convention is to make your iterator variable to be singular. So here I'm going to say for player in player's come down here and say console log players and then player.

Now one important thing to note here, player doesn't actually represent the value. Player represents the index. We can see that here if we want. So here let me do another console log. Just so he can see exactly what is what player represents so clear.

```
for (player in players) {
console.log(player);
console.log(players[player]);
}
```

If I run this you can see we have Altuve and zero. Now we have 1 followed by Bregman. So what this is doing. It's very similar to our original for loop that we put together. The only difference is we don't have to put in our own conditions. We also don't have to have an incrementor, the reason for that is because for-in automatically does that work for you because it can see that we are only going to loop as many times over as many elements that are contained inside of the collection.

This is something that is definitely a little bit more modern. You will see this in quite a few of the applications in the frameworks that are out there now for loops are still very handy to work with especially if you're working with non-collection data.

There are many times where I will have to work with something that is not an easy array like this instead it's something a little bit more abstract or more dynamic and I have to control the parameters to be a little bit more strict. Whereas here what this does is it relies on the understanding that we only care about iterating over a collection as many times and for as many elements as are in the collection.

This is something I would do if I were to be iterating over say a database query. That's something that's a little bit more common. And I'm also going to walk through another example and it's going to be our last one for this guide.

For has another option which is called the For Each loop. So we have our FOR loop. Now let's talk about our For Each loop.

I call this players so say players and forEach. Inside of this, we have a number of items that we put inside of it. This gets more into how we can pass functions around.

```
players.forEach(function(element) {
console.log(element);
});
```

So after we've called for each on players we're going to pass in a function. So here is a function and it takes an argument but we're technically not going to be passing an argument in because all of this happens automatically and I'll show you what that means here in a second. So now I can say console log and then the element and then let's close this off and hit clear.

If I hit run now you can see this does exactly the same thing. So for each is something that is relatively new to javascript for years we just had for and then for in came into play. And both of those were fantastic for in worked very nicely with collections but now probably one the most modern things you will see is a For Each loop.

If you notice this takes a very similar pattern to the rest of the code that we've been writing. This looks much more functional. This is a function being called on a collection. This fits a little bit more with a modern javascript programming paradigm. And so you're going to see this a lot. This is one of the ones I use quite a bit now just because I really like the syntax it's in very nice but also because it is truly functional. it gives me the ability to place this inside of say an object or placing this inside of another function I want to be able to iterate through.

As you get into working with a framework such as angular or react you're going to see this type of design pattern quite a bit.

So in review, we just covered three different types of for loops we covered a traditional for loop. We covered a for-in loop and lastly, we covered the more functional version of it which is the For Each loop.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];

for (player in players) {
  console.log(players[player]);
}

for (var i = 0; i < players.length; i++) {
  console.log(players[i]);
```

```
}

players.forEach(function(element) {
  console.log(element);
});
```

Coding Exercise

Create an array called "members" with 5 elements. Write a traditional for loop that uses an iterator and iterates through the array and console logs each member

NOTES:

```
/*
Visualize the logic inside a function.

function iterateByMembers() {
*/
    var members = ['user1','user2','user3','user4','user5'];
// typical for loop
    for ( var i = 0; i < members.length; i++ ) { // try to write all
the iter. conditions in one sentence
        console.log(members[i]);
    }
    /*
}
*/
```

5.3 How to Loop Through a JavaScript Object

This lesson examines how you can leverage the for-in loop in order to loop over a JavaScript Object.

THEORY:

Iterating Over Objects

The `for...in` loop is a useful tool for iterating over the keys (properties) of a JavaScript object.

Example: Student Object

```
var student = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale'
};

for (var key in student) {
  console.log(key + " => " + student[key]);
}
```

Explanation

- `for (var key in student)`: This loop iterates over each property in the `student` object. In each iteration, the `key` variable is assigned the name of the property.
- `console.log(key + " => " + student[key]);`: This line prints the property name (`key`) followed by an arrow (`=>`) and the value of the property (`student[key]`).

Accessing Object Properties

There are two ways to access object properties in JavaScript:

1. **Dot Notation:** `object.property`
2. **Bracket Notation:** `object['property']`

Why Bracket Notation is Used in the Loop

In the `for...in` loop, the `key` variable holds a string representing the property name. To dynamically access the property value using this string, you need to use bracket notation (`student[key]`). Dot notation (`student.key`) would try to access a property literally named "key," which doesn't exist in this case.

Example: Product Object

```
let product = {
  name: 'T-shirt',
  price: 19.99,
  size: 'M',
  color: 'Blue'
};

for (let property in product) {
  console.log(property + " => " + product[property]);
}
```

Output

```
name => T-shirt price => 19.99 size => M color => Blue
```

VIDEO:

In the last guide, we talked about how we could use various for loops to be able to iterate over a collection of data. In this guide, we're going to talk about how we can do a similar technique to loop over an object.

This is something that is very common, Imagine an API call that you're making. In other words, you go out to another server say it's a service like Twitter and you pull down tweets. While those tweets are sent in many of the components are sent in an object format where they have a key-value pair kind of structure.

If you want to show that on the page then you're going to have to know how to iterate over those. One of the most common ways is to iterate over an object in javascript is to use the "for in loop" that we already walkthrough.

In order to do that we have an object here, it's a student with name age and city.

To iterate over this and print out each one of the items I'm going to say for and then var key in student. This is something in the last guide, I forgot to put a var in front of key. It still worked but it would be considered a bit of an anti-pattern because essentially I would be polluting the global namespace. Remember that whenever you just declare a variable, in this case, we just assigned it, then it would get put in the global namespace where if I put it and say var key it's going to be local to this block.

```
var student = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale'
};
```

```
for (var key in student) {
  console.log(key + " => " + student[key]);
}
```

So I'm going to say var key in student. Now inside of this, I want to print out and say key plus and then I'm going to do a little hash rocket sign here. And so an equals. You could have it do this it's completely up to you just so it can say which key is associated with which value.

Inside of this I'll say student and using the bracket notation which you know that's how you can go and grab an object. You can grab a value by the key by calling the object and passing in with brackets whatever the key is. If I pass in the key name it will pull it in. I want to show you this syntax in addition to the dot syntax that we've used up to this point.

Now if I hit run this is going to run and this worked. So I have name points to Kristine age 12 city Scottsdale so all of that is working properly.

Now if I come back here and say student.key

```
var student = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale'
};

for (var key in student) {
  console.log(key + " => " + student.key);
}
```

hit run everything still works on the key side but not on the value side and this is a reason why I wanted to show you the second type of syntax because up until this entire time we've only used the key-value pair syntax and then we used the dot syntax.

So what we've done is instead of just being able to call name or age with a dot the key doesn't get rendered quite like that. So our key is going to be processed differently than when it's like a function or an attribute call on here, it's not going to work that same way.

We have to be able to have a secondary syntax for it and that's where we use the brackets. So in this case this is the correct syntax. Many times and I would say the majority of the time you're going to do something like student.name and now work perfectly fine and that's considered the best practice but in certain cases wherein this case key is as much a true key as it is just an iterator variable. Then we need to be a little bit more explicit with it in use this bracket syntax.

So that is how you can loop over a collection that is an object using the for-in loop in javascript.

```
var student = {
  name: 'Kristine',
  age: 12,
  city: 'Scottsdale'
};

for (var key in student) {
  console.log(key + " => " + student[key]);
}
```

Coding Exercise

Create an object called "user". Assign it a username, email, phone and give them values. Console log the data in the same format you can see here:

Example:

```
let producto = {
    nombre: 'Camiseta',
    precio: 19.99,
    talla: 'M',
    color: 'Azul'
};

for (let propiedad in producto) {
    console.log("La propiedad " + propiedad + " tiene el valor: " +
producto[propiedad]);
}

let user = {

};
```

5.4 While and Do/While Loops

This lesson examines how you can work with Do and Do/While loops in JavaScript programs.

THEORY:

`while` Loop

The `while` loop repeatedly executes a block of code as long as a condition is true.

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Explanation

1. **Initialization:** A counter variable (`i`) is initialized before the loop.
2. **Condition:** The `while` loop checks the condition (`i < 5`).
3. **Execution:** If the condition is true, the code block is executed.
4. **Iteration:** The counter variable is updated (`i++`), and the loop repeats from step 2.
5. **Termination:** The loop terminates when the condition becomes false.

`do...while` Loop

The `do...while` loop is similar to the `while` loop, but it executes the code block at least once before checking the condition

VIDEO:

So far we've covered a number of the loop types, everything from "for" all the way through the functional "for each." We also saw how we can iterate over an object.

I want to talk about two more loops. These loops are nowhere near as popular as the "for" variety that we've already discussed. However, it would be a bad job on my part if I didn't tell you about them because there are still times where they can be useful and also if you're working on legacy

applications you're more likely to run into these at some point or another because some developers prefer them.

They're not my personal preference so you won't find them in very many of the applications that I make but they still serve a purpose and I am going to point out one specific scenario later on which this is a perfect fit for a looping mechanism.

First, let's talk about the "while" and then the "do while" loop. First, we need to declare an iterator variable we can call it anything that we want. We also have our array up here. While loops do not have the same type of setup that a for loop has even a traditional for loop where you can put the variable inside of the function expression itself. Now with loops, we have to declare it outside.

So first and foremost that is a good thing but it also can be about things. It might be an issue where we're having to declare a variable outside of the scope of the loop which means we have access to it even afterward, that may or may not be something that you want.

So now that we've declared our variable let's create the while loop.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];

var i = 0;
while (i < players.length) {
  console.log(players[i]);
  i++;
}
```

So you say while i is less than players.length, this is pretty similar to the conditional that we put in the for loop, I want to console log out players and then grab the index because i's functioning as an index. Now it's very important we do not run this right now and the reason is that this will throw what's called an infinite loop.

If you've noticed we don't have anything that changes the value of i yet when we had our traditional for loop if you remember the very last item on the list was an incrementor, just like the same way where we had to assign a value to i in this case we have to also manually create our own incrementor. Inside of the While loop. I'm going to say i ++ and now this will work.

Every time it comes through the loop it's going to increase it's going to come back up here it's going to check to see is i less than the players length which in this case we know is 4. If so it's going to keep going as soon as it's equal to or greater than it's going to stop the execution.

If I hit run this runs and it works we get all 4 of our players printed out. So that is good.

That is a traditional while loop, it has a close cousin called the "do while" loop, and the syntax for that is very similar except almost in reverse.

You take out this entire while portion and you put it at the very end of your expression or out of your block. At the top where while was now, you say do and this is why it's called a do-while loop.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];
```

```
var i = 21;
do {
  console.log(players[i]);
  i++;
} while (i < players.length)
```

When you have do and then while right here it is going to give you a slightly different set of behavior. It is because with a do loop the condition is checked after the iteration, with a while loop it's checked first.

That may not seem like a big deal and if I hit clear right here and then run it everything works identically in this scenario and that's perfectly fine. However, let's imagine a scenario where I set i equal to 21.

If I hit clear and hit run only one thing gets to print out and it's undefined. Now if I would have done that for the while loop nothing would have been printed out and just so you know I'm not lying to you. Let me do it right now. So now if I do the same thing with my while loop and run nothing gets printed out and that is the main difference is that with a while loop it does its conditional check first. With a do-while loop, it is always going to go through the program at least one time.

You may think that that sounds weird and that you would never have a reason to use a do-while loop, actually, I have through the years found myself needing to use a do-while loop even more than a while loop. Mainly because I never really use a while loop because for loops work much better. However a do-while loop can do something that none of the other loops can do, that is it can perform and guarantee that it will run at least one time.

Imagine that you're building a game if you're building a game and you always want one process to run at least once you want to have that guarantee then your game can use a do-while loop and in fact, that is one of the most common times that I see do while loops used is in games where you want a process that runs at least once.

Say that you have a player who's playing your game and you want to be guaranteed that the program is going to run at least one time then if they keep on winning then it can check that in the while condition and it can keep on looping through. You always have that one guaranteed one in the beginning.

That's one of the more common spots where you'll see a do while implemented. In review, we talked about two different types of loops talked about while and do while. We talked about their positives negatives what you have to do from a syntax perspective and then finished off with a practical example on why you'd want to use a do-while loop in a game type of scenario.

```
var players = [
  'Altuve',
  'Bregman',
  'Correa',
  'Springer'
];

var i = 0;
while (i < players.length) {
  console.log(players[i]);
  i++;
}

var i = 21;
do {
```

```
  console.log(players[i]);
  i++;
} while (i < players.length)
```

Coding Exercise

In the below starter code, place 4 dog names (elements) of your choice. Write a while loop that iterates through the `dogHouse` array. The iterator variable must be named "data".

```
//Please leave the below starter code but fill in the array with 4
elements
var dogHouse = [];
```

MODULE 6.

AUTOMATING TASKS

6.1 Section Introduction: Introduction to Automating Tasks in the Browser with JavaScript

Hi, and welcome to this section of the course where we're going to learn how we can automate tasks in the browser using JavaScript.

Remember that every `element` in the browser, such as `links`, `headings`, `images`, those are all included in what is called the **`DOM` or the `document object model`**. Because of that, it means that JavaScript is able to work with every element on the page just like any other object.

That's what the goal of this section the course is going to be is to show you how you can take your JavaScript code and how you can traverse the DOM. We're going to use some great case studies such as LinkedIn, Google, and Instagram for seeing how we can have our own unique JavaScript code that will go through a page and it will perform a task as many times as you need.

For example, if you want to follow a bunch of people or hashtags on LinkedIn, then you can find the `class` or the `element` on the page that has that button. Then we're going to walk through how you can click all of the elements on the page by just running a single command. There's all kinds of other `automation tasks` that we can use for connecting JavaScript with the browser.

6.2 Running Google Search Queries in JavaScript

By understanding these techniques, you can automate various tasks on web pages and build more dynamic and interactive applications.

THEORY:

Understanding the Task

The goal is to write a JavaScript snippet that automatically:

1. Enters a search query into the Google Search bar.
2. Clicks the "Google Search" button.

Inspecting the Page

To interact with web page elements using JavaScript, you need to understand their structure. Use your browser's developer tools (right-click and select "Inspect") to examine the HTML of the Google Search page.

Identifying Key Elements

- **Search Bar:** Locate the HTML element for the search bar. Note its `class` or `id` attribute, which you'll use to target it with JavaScript.
- **Search Button:** Similarly, find the "Google Search" button element and note its `class` or `id`.

Writing the JavaScript Code

```
// Enter the search query
document.querySelector('.gLFyf').value = "Javascript tips";

// Click the search button (updated selector)
document.querySelector(".FPdoLc").childNodes[1].childNodes[1].click();
```

Explanation

- `document.querySelector('.gLFyf')`: Selects the search bar element using its class name.

- `.value = "Javascript tips";`: Sets the value of the search bar to the desired query.
- `document.querySelector(".FPdoLc")`: Selects an ancestor element of the search button.
- `.childNodes[1].childNodes[1]`: Traverses the DOM to locate the specific button element.
- `.click()`: Simulates a click on the search button.

Important Notes

- **Website Updates:** Google's website structure might change, requiring you to update the selectors in your code.
- **DOM Traversal:** If the search button doesn't have a unique `class` or `id`, you might need to traverse the DOM (using `childNodes`, `parentNode`, etc.) to locate it.
- **Alternative Approach (jQuery):** If you're using jQuery, you can use its simpler syntax for selecting elements and performing actions.

Example (jQuery)

```
$('.gLFyf').value = "Javascript tips";
$(".FPdoLc").childNodes[1].childNodes[1].click();
```

Important:

- You can automate web page interactions using JavaScript.
- Inspect the page's HTML to identify the elements you want to interact with.
- Use `document.querySelector` or jQuery selectors to target elements.
- Simulate actions like entering text and clicking buttons using JavaScript.

VIDEO:

Code Update

Google has updated its website, this is the way to call its new classNames.

Instead of running `$('.gsfi').value = 'JavaScript tips'` to input text into the search bar the updated command is `$('.gLFyf').value = "Javascript tips";`

And to perform a search `$('.jsb').childNodes[0].childNodes[0].click` has been updated to `$(".FPdoLc").childNodes[1].childNodes[1].click()`

Now one caveat I will say in this section is: I'm going to start with the first few videos teaching you more the fundamentals. We are going to build out these scripts, but they're not going to probably be things that you'll be doing on a daily basis.

The reason for that is because I think it's important to understand exactly what's going on before we get to the more challenging topic. We will eventually get to LinkedIn and we'll see how we can automate the ability to follow a bunch of hashtags all at one time. That kind of thing.

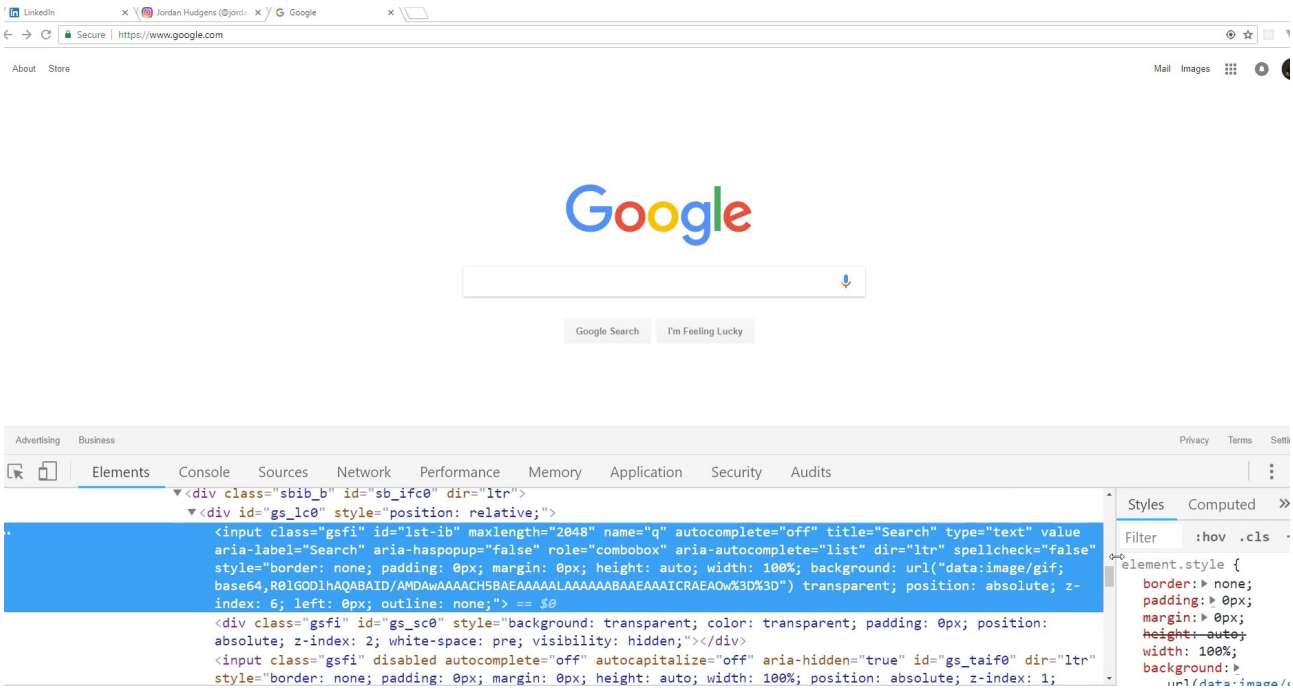
You could follow along with that easily by just typing out the code I type. I think if we take a few easier examples in the beginning, then whenever you need to create your own script completely from scratch, you'll have a lot better idea of how I went about it, as opposed if I jumped to the most challenging one right away.

With all that being said, the goal is to create a script that types into this little search bar right here, and then hits enter. So this would be the same as if I did something like this so I said `some query`, and then clicked on google search. That is what I want to automate. I want to have this filled in and then I want to click on google search.

Any time that I'm building out a new script like this, the very first thing that I do is I just try to figure out what kind of data I'm working with because that's the first step. Because we're first going to have to select the value, we're going to have to traverse through this `HTML document`. We're going to have to find the value we want and then we can work with that.

If I were to `right-click` here, and click on `view page source`: this is all the code that makes up that Google front-end. Now that isn't very fun to go through. So what I do instead is I right-click and then click on `inspect`. You could also click `control + shift + i`.

I like to do that so that shows me exactly what I'm working with, and I'm going to slide this to the right. So it's a little bit easier to see.

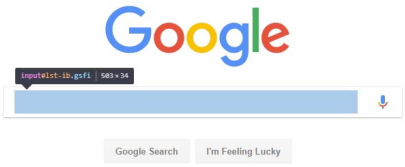


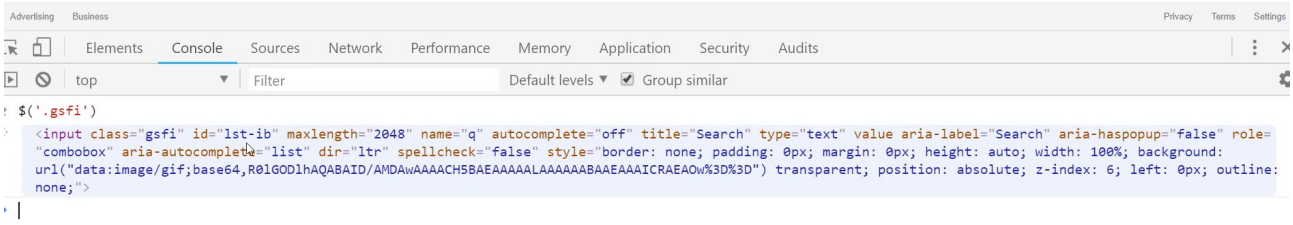
Now what this tells me is it tells me some details about the `input element`. Thankfully this element has a `class` in an `ID`. Those are two things that we could use and we could work with. That is the first thing that I know that we need, and this is going to be the easy part is actually setting the text. Then, later on, we're going to see how we can click the button.

First, let's just make sure that we have a script so that we can do this. I'm going to use the class, but we also could use the ID. The only difference is, remember that an ID can only be placed one spot on the page and a class is you could put as many classes as you want.

In this specific case, it doesn't matter because I know there is only one of these on the page. So I'm going to click here and I'm going to show you two ways that you can use a `selector` inside of a browser.

The first is to use this `$` syntax where I say `$` and then in parentheses, I go with a quote, and then I can just paste in the className that we grab. Now if I finish that off and hit enter you can see it returns that entire `DOM node`, which is this `input element` right there.

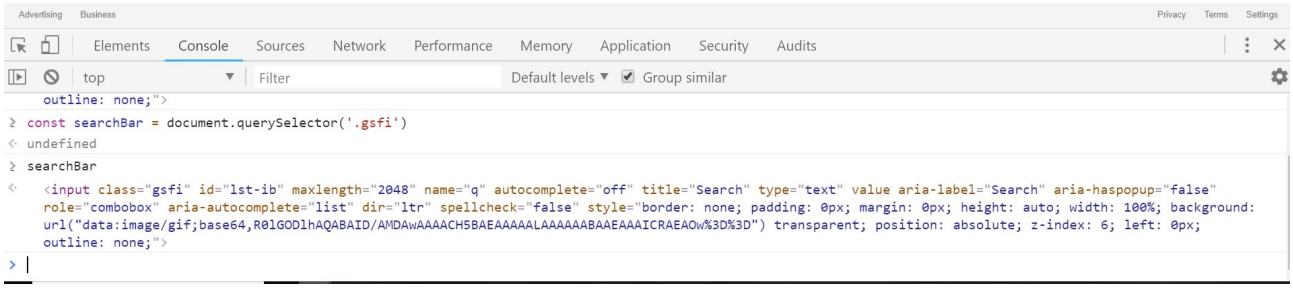




That is perfectly fine. Now the other way that could do this is writing pure Javascript. So I could say something like:

```
const searchBar = document.querySelector('.gsfi')
```

In this case, I only want to grab one so I can say `querySelector` and not `querySelectorAll`. Then use the exact same syntax and it will say `undefined`, but then if I find my search bar you can see it returns the exact same value.



So everything that we did with that first syntax can be accomplished in regular Javascript. Now the key difference here is if you're building out in testing a real javascript code snippet, something that you're going to put in a javascript file, then you have to go with this syntax.

If you simply want to automate something in the browser then you can use this. This gives you a really nice and easy API for simply grabbing elements and then working with them. This video I'm going to use this syntax, and then in future videos, I'm going to show you how to use the other one.

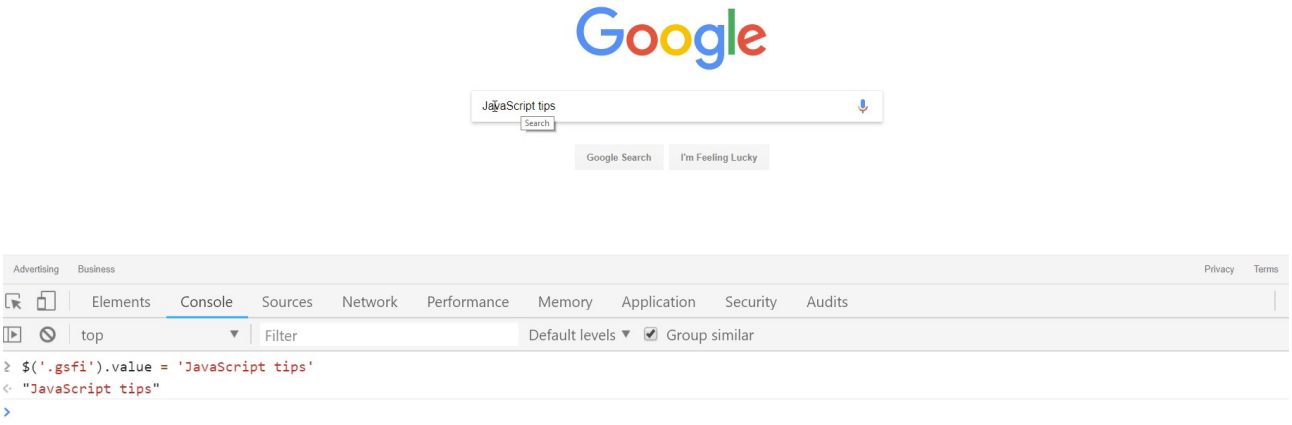
Let me just clear this off, and you can clear it by pressing `control + l`. Now what I want to do is I want to set the value, so I'm going to say: this class and then I'll say:

```
$('.gsfi').value = ''
```

Then let's just have a string. I'll say I want to search for JavaScript tips just like this:

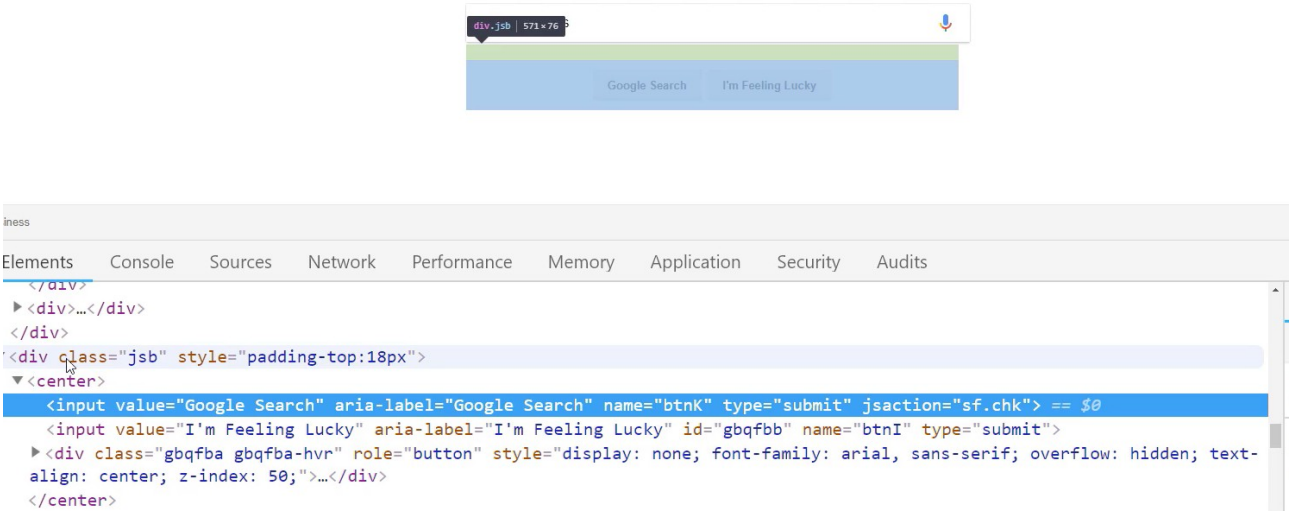
```
$('.gsfi').value = 'JavaScript tips'
```

You can see right up here. It now says javascript tips.

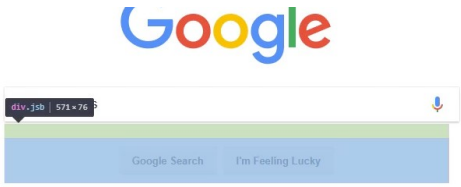


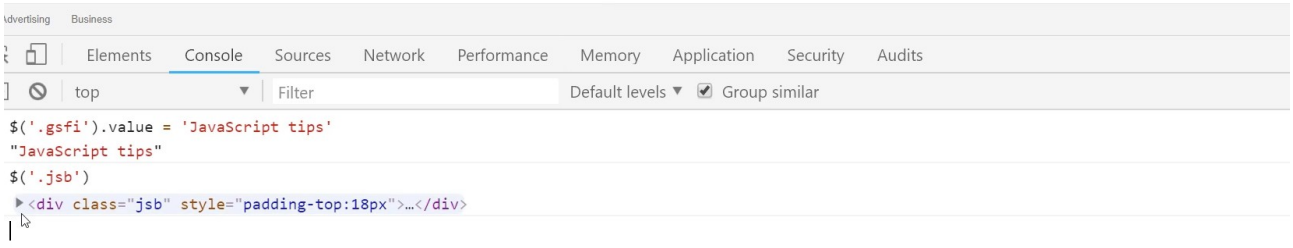
Now there is only one little trick to this, and that is: do you notice how there is not an ID or a className here? That means that we can't use our normal selectors. What we're going to have to do is we're going to have to it's called `traversing the DOM`, which means we're going to grab the element a little bit higher up.

In fact, we're going to grab this `JSB class element`, and then we are going to search through the elements until we find the Google search ones.



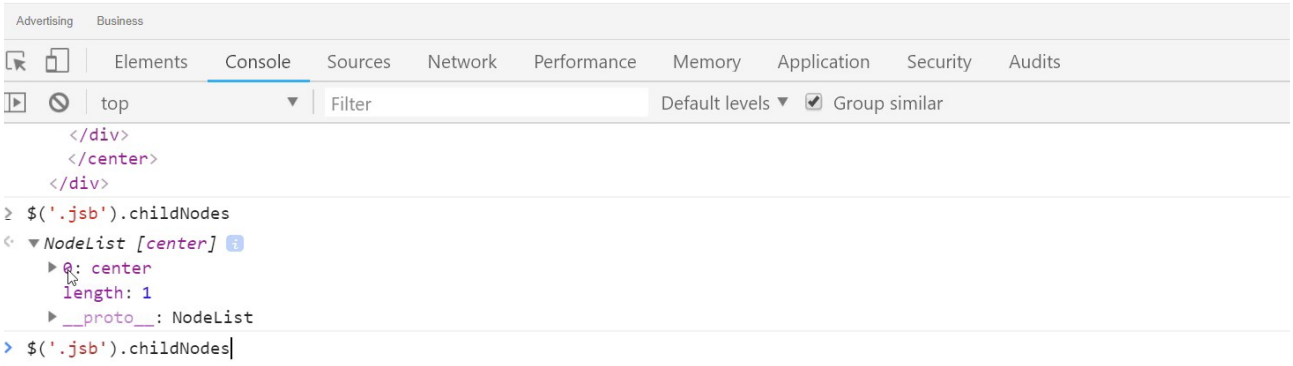
I'm just going to copy this JSB one here, and I'm going to use the same syntax. I'm going to say: `$('.jsb')` just like that. You can see that gives us the entire object. This is good, but we don't really care about clicking on this wrapper. As you can see when I hover over here that is highlighting everything.





So we want to actually get only this Google search button. If you click here, you can see that button is inside of `center`, and that is where we can find it. So what I'm going to do now is I'm going to hit `up`. I can say: `$('.jsb')`, and now the function that I want to run is called `childNodes`.

What this tells JavaScript and what it does in the browser is it says: "okay, thank you for bringing me this JSB here, but I'm actually interested in what's inside of it. I'm interested in what's nested there." So now if I click on that, you can see it returns a `NodeList`, and this NodeList has a single element called `center`.

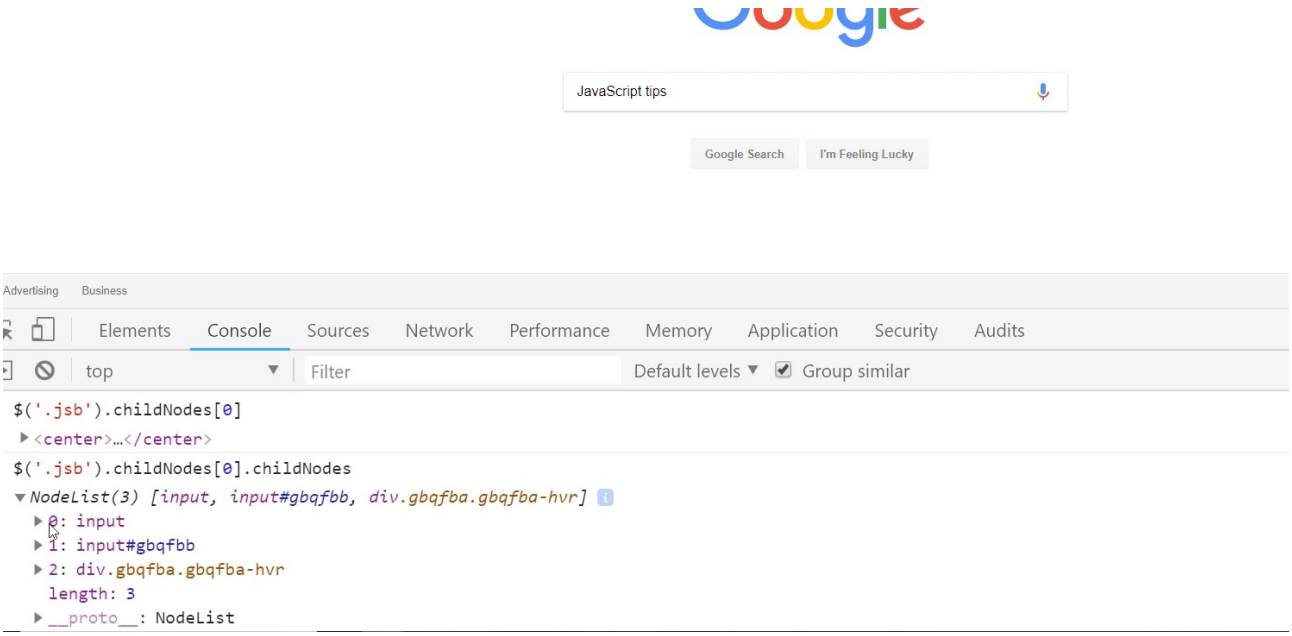


That's fine. We want to grab this element, we can see it has an index of `0`, so here I can say `$('.jsb').childNodes[0]`, and let's just treat this as an array. It's not technically an array, it's in a NodeList, but we can use this bracket syntax. Now we have our center object.

Do you notice how if you come back here, you know how we started at JSB then we moved to center, so now it's like we've moved one level down? Now what we can do is see what childNodes center has. So I can say:

```
$('.jsb').childNodes[0].childNodes
```

Now you can see we're getting closer.



Now we have these three inputs here, and this first one is what we're looking for. This is the Google search. So in order to get what we actually want, let's just hit `up` once, and because that's a node list we can treat it very similar to an array. Now if I give `[0]`, we have exactly what we're looking for.

We finally have that Google search, and now all we have to do is call the click function. So I can say:

```
$('.jsb').childNodes[0].childNodes[0].click6.2
```

6.3 Running Google Search Queries in JavaScript

This guide demonstrates how to automate Google Search queries using JavaScript. It appears you've provided the transcript from a video tutorial, which is quite helpful! I'll break down the process, incorporating the updates you mentioned.

THEORY:

1. Inspect the Page

- Open the Google Search page in your browser.
- Right-click and select "Inspect" to open the developer tools.

2. Identify Key Elements

- **Search Bar:** Find the search bar element. Its `class` attribute should be `gLFyf`.
- **Search Button:** The search button might be nested. You'll likely need to traverse the DOM starting from an ancestor element. In your example, you start with the element with class `FPdoLc`.

3. Write the JavaScript

```
// Enter the search query (updated selector)
document.querySelector('.gLFyf').value = "Javascript tips";

// Click the search button (updated selector)
document.querySelector(".FPdoLc").childNodes[1].childNodes[1].click();
```

Explanation

- `document.querySelector('.gLFyf')`: This line targets the search bar element using its class name.
- `.value = "Javascript tips";`: This sets the value of the search bar, effectively entering the text.
- `document.querySelector(".FPdoLc")`: This targets an ancestor element of the search button.
- `.childNodes[1].childNodes[1]`: This navigates through the child nodes of the ancestor element to reach the actual search button.
- `.click()`: This simulates a click on the button, initiating the search.

Important Notes

- **Google's DOM Changes:** Google frequently updates its page structure. The selectors used in this example might need adjustments if Google changes its HTML. Always inspect the page and adjust your selectors accordingly.

- **jQuery:** The example you provided uses jQuery (indicated by the `$`). While jQuery can simplify DOM manipulation, the core concepts remain the same.

VIDEO:

Code Update

Google has updated its website, this is the way to call its new classNames.

Instead of running `$('.gsfi').value = 'JavaScript tips'` to input text into the search bar the updated command is `$('.gLFyf').value = "Javascript tips";`

And to perform a search `$('.jsb').childNodes[0].childNodes[0].click` has been updated to `$(".FPdoLc").childNodes[1].childNodes[1].click()`

Now one caveat I will say in this section is: I'm going to start with the first few videos teaching you more the fundamentals. We are going to build out these scripts, but they're not going to probably be things that you'll be doing on a daily basis.

The reason for that is because I think it's important to understand exactly what's going on before we get to the more challenging topic. We will eventually get to LinkedIn and we'll see how we can automate the ability to follow a bunch of hashtags all at one time. That kind of thing.

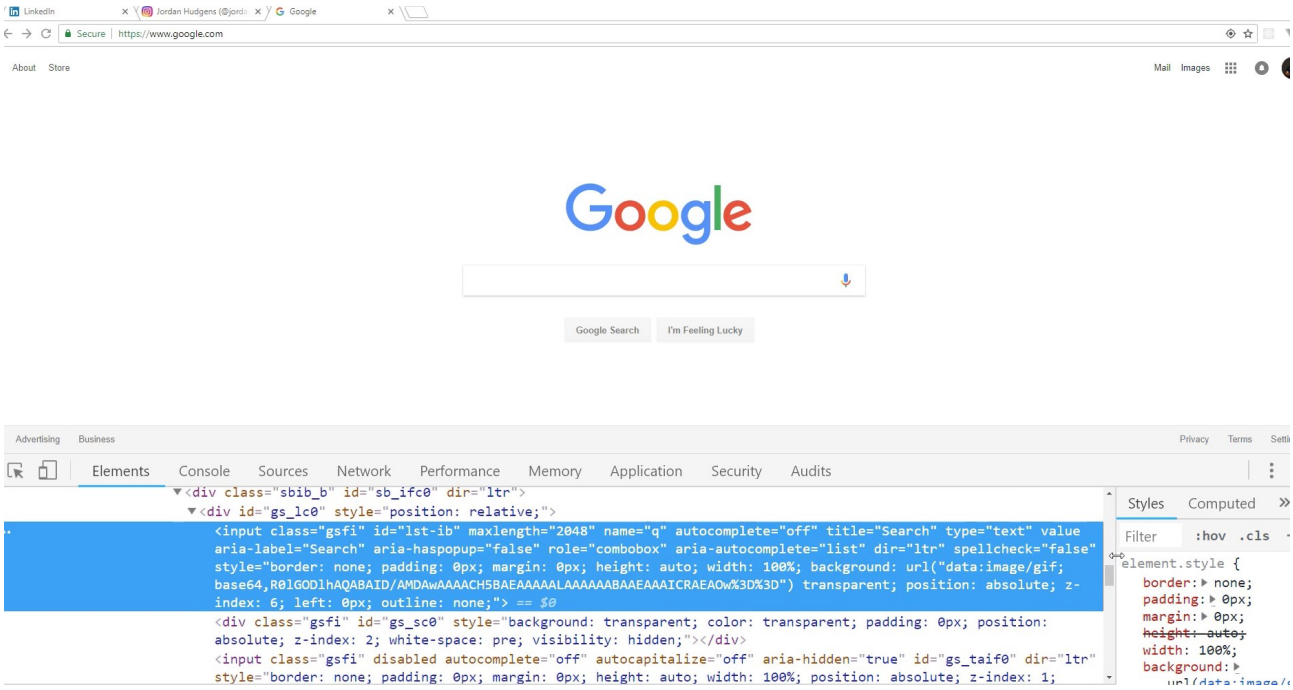
You could follow along with that easily by just typing out the code I type. I think if we take a few easier examples in the beginning, then whenever you need to create your own script completely from scratch, you'll have a lot better idea of how I went about it, as opposed if I jumped to the most challenging one right away.

With all that being said, the goal is to create a script that types into this little search bar right here, and then hits enter. So this would be the same as if I did something like this so I said `some query`, and then clicked on google search. That is what I want to automate. I want to have this filled in and then I want to click on google search.

Any time that I'm building out a new script like this, the very first thing that I do is I just try to figure out what kind of data I'm working with because that's the first step. Because we're first going to have to select the value, we're going to have to traverse through this `HTML document`. We're going to have to find the value we want and then we can work with that.

If I were to `right-click` here, and click on `view page source`: this is all the code that makes up that Google front-end. Now that isn't very fun to go through. So what I do instead is I right-click and then click on `inspect`. You could also click `control + shift + i`.

I like to do that so that shows me exactly what I'm working with, and I'm going to slide this to the right. So it's a little bit easier to see.

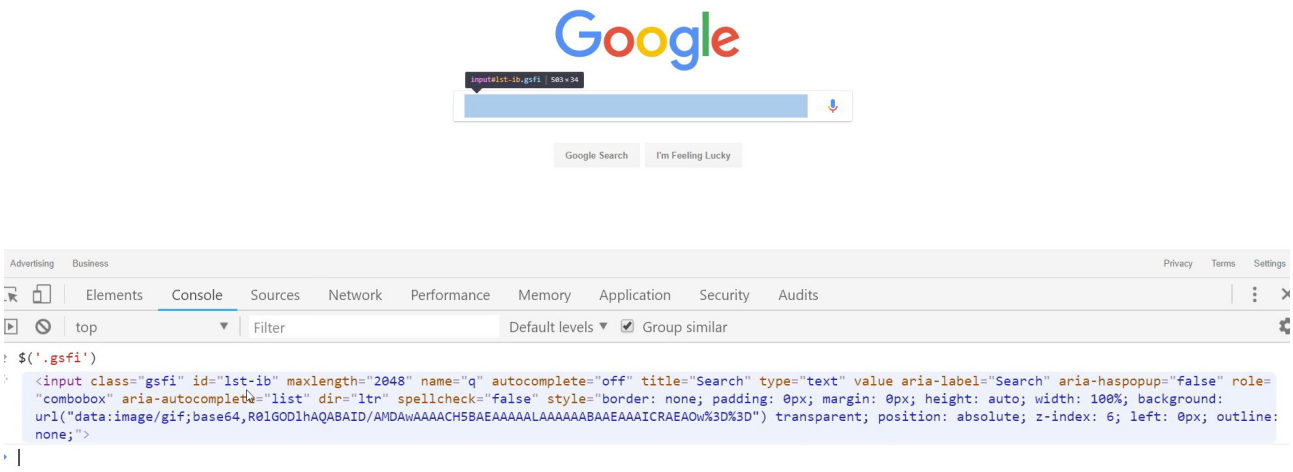


Now what this tells me is it tells me some details about the `input element`. Thankfully this element has a `class` in an `ID`. Those are two things that we could use and we could work with. That is the first thing that I know that we need, and this is going to be the easy part is actually setting the text. Then, later on, we're going to see how we can click the button.

First, let's just make sure that we have a script so that we can do this. I'm going to use the class, but we also could use the ID. The only difference is, remember that an ID can only be placed one spot on the page and a class is you could put as many classes as you want.

In this specific case, it doesn't matter because I know there is only one of these on the page. So I'm going to click here and I'm going to show you two ways that you can use a `selector` inside of a browser.

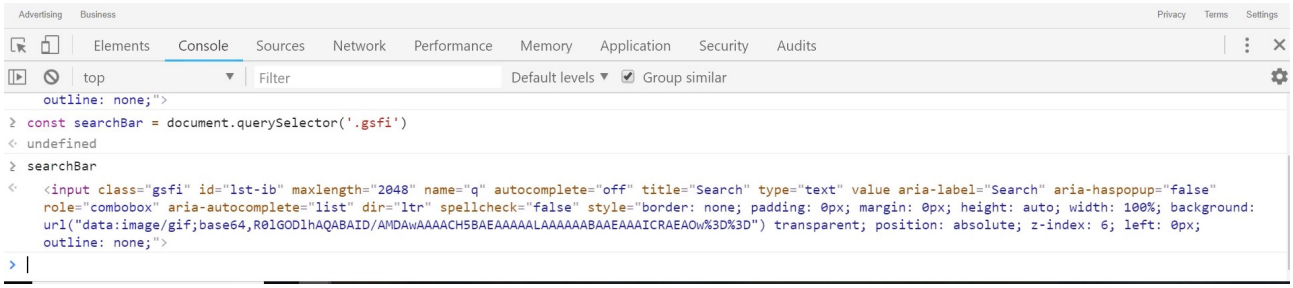
The first is to use this `$` syntax where I say `$` and then in parentheses, I go with a quote, and then I can just paste in the className that we grab. Now if I finish that off and hit enter you can see it returns that entire `DOM node`, which is this `input element` right there.



That is perfectly fine. Now the other way that could do this is writing pure Javascript. So I could say something like:

```
const searchBar = document.querySelector('.gsfi')
```

In this case, I only want to grab one so I can say `querySelector` and not `querySelectorAll`. Then use the exact same syntax and it will say `undefined`, but then if I find my search bar you can see it returns the exact same value.



So everything that we did with that first syntax can be accomplished in regular Javascript. Now the key difference here is if you're building out in testing a real javascript code snippet, something that you're going to put in a javascript file, then you have to go with this syntax.

If you simply want to automate something in the browser then you can use this. This gives you a really nice and easy API for simply grabbing elements and then working with them. This video I'm going to use this syntax, and then in future videos, I'm going to show you how to use the other one.

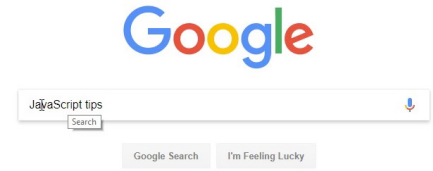
Let me just clear this off, and you can clear it by pressing `control + l`. Now what I want to do is I want to set the value, so I'm going to say: this class and then I'll say:

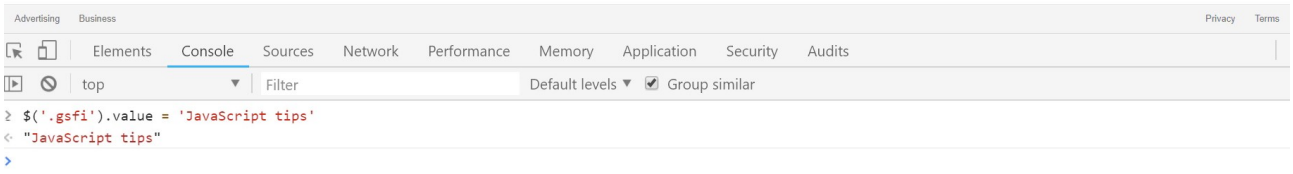
```
$('.gsfi').value = ''
```

Then let's just have a string. I'll say I want to search for JavaScript tips just like this:

```
$('.gsfi').value = 'JavaScript tips'
```

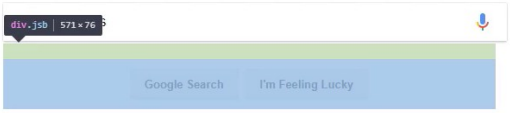
You can see right up here. It now says javascript tips.

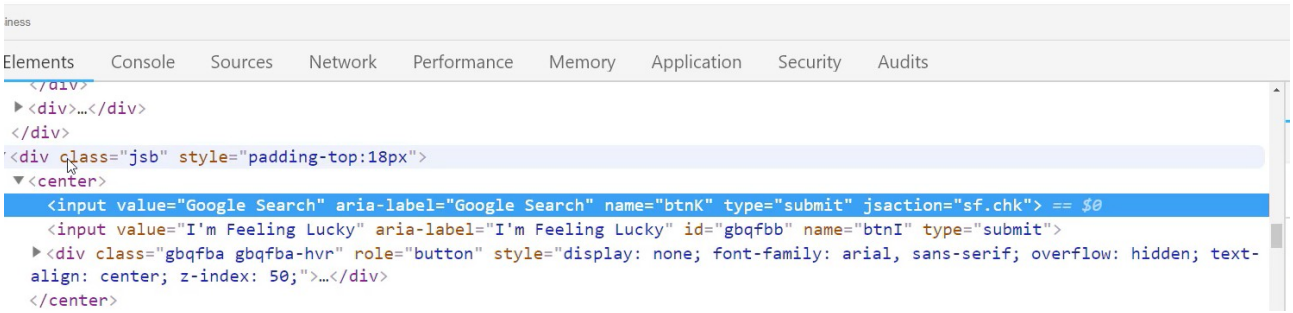




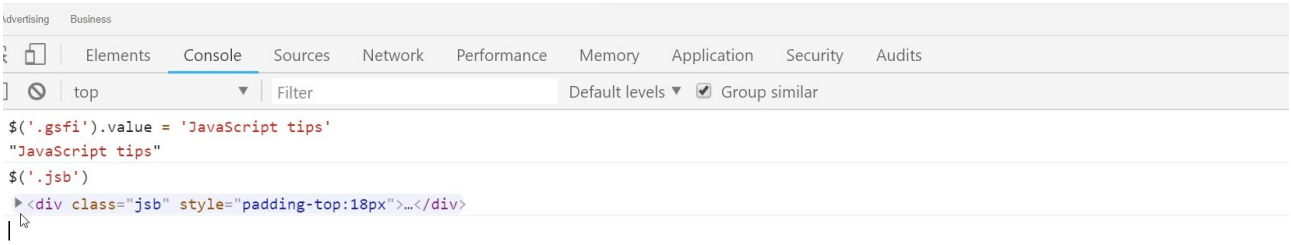
Now there is only one little trick to this, and that is: do you notice how there is not an ID or a className here? That means that we can't use our normal selectors. What we're going to have to do is we're going to have to it's called `traversing the DOM`, which means we're going to grab the element a little bit higher up.

In fact, we're going to grab this `JSB class element`, and then we are going to search through the elements until we find the Google search ones.



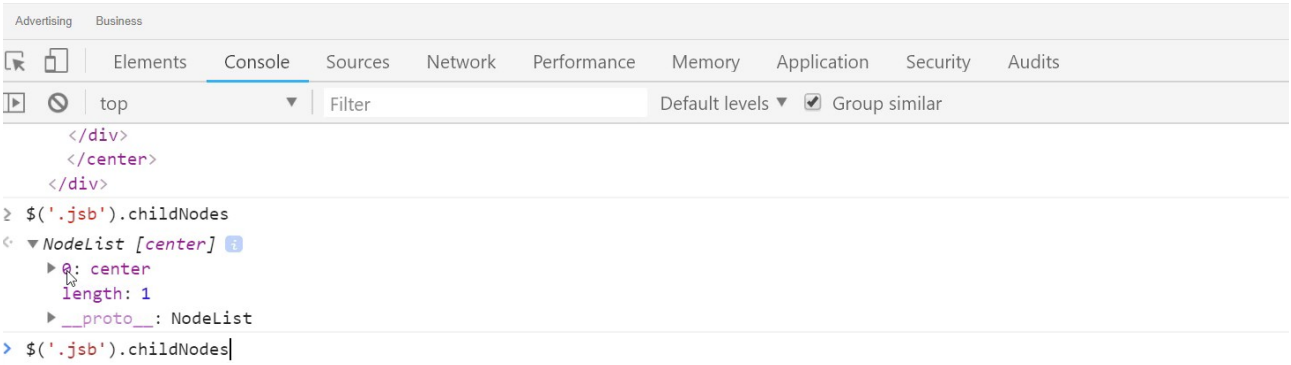


I'm just going to copy this JSB one here, and I'm going to use the same syntax. I'm going to say: `$('.jsb')` just like that. You can see that gives us the entire object. This is good, but we don't really care about clicking on this wrapper. As you can see when I hover over here that is highlighting everything.



So we want to actually get only this Google search button. If you click here, you can see that button is inside of `center`, and that is where we can find it. So what I'm going to do now is I'm going to hit `up`. I can say: `$('.jsb')`, and now the function that I want to run is called `childNodes`.

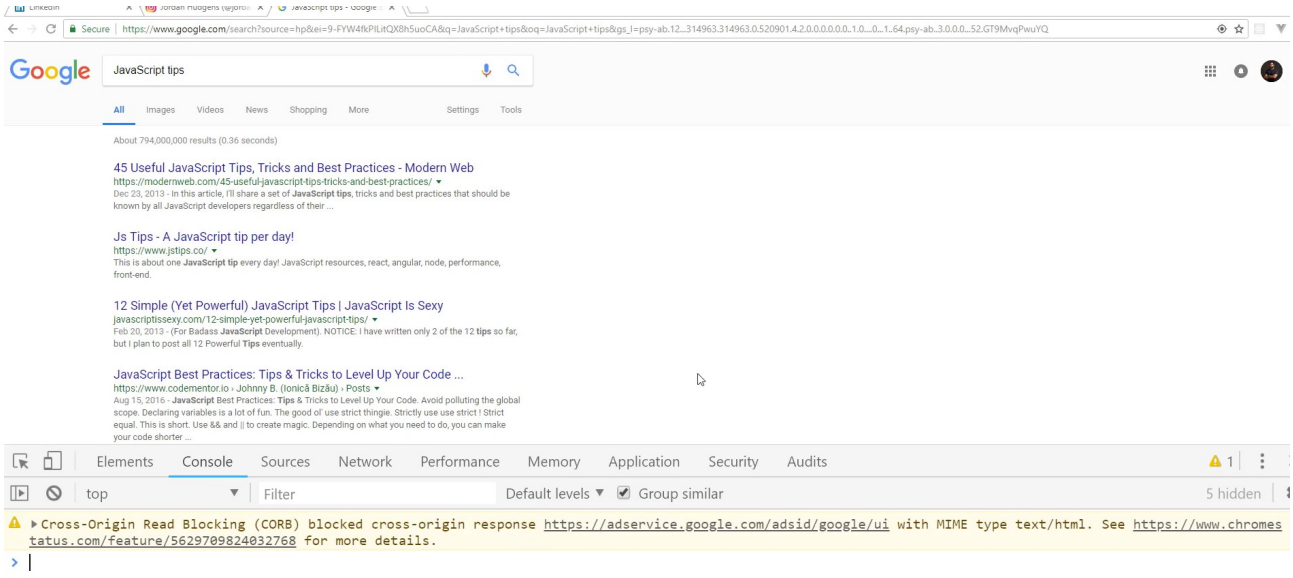
What this tells JavaScript and what it does in the browser is it says: "okay, thank you for bringing me this JSB here, but I'm actually interested in what's inside of it. I'm interested in what's nested there." So now if I click on that, you can see it returns a `NodeList`, and this NodeList has a single element called `center`.



That's fine. We want to grab this element, we can see it has an index of 0, so here I can say $('.jsb').childNodes[0], and let's just treat this as

What javascript is going to do is it actually going to, it still has our javascript tips text here, and then it's going to call click on whatever element we've selected which is this button.

Now if I hit enter. It clicked, it worked, and you notice we didn't actually have to do anything ourselves. Now it searched for JavaScript tips, so that is working perfectly. Now like I said at the beginning this is not an automation tool that you will probably be using directly, because usually it's just a lot easier to type into the search bar.



Hopefully, now you have a little bit more of an idea on first how you can find the elements that you want to select in the DOM in the browser. Then from there how you can traverse to elements you can't quite select as much along with being able to use functions such as selecting and then clicking. We'll see you in the next video where we're going to take a look at Instagram.

Coding Exercise

Use the querySelector to grab the div with the class name "target"

```
<div class="parent">
    <div class="decoy"></div>
    <div class="decoy"></div>
    <div class="target">You got this!</div>
    <div class="decoy"></div>
</div>
var query = deleteThisAndReplaceWithYourCode;
```

6.3 How to Pull Images from Instagram with JavaScript

This is going to be a fun guide, especially if you are a fan of Instagram. What we're going to walk through here is how to go and grab all of the Instagram URLs from a page.

THEORY:

1. Inspect the Page

- Open your Instagram profile page.
- Right-click on an image and select "Inspect" to open the developer tools.
- Identify the HTML element that holds the image. In your example, it has the class `FFVAD`.

2. Select the Images

- Use `document.querySelectorAll()` to select all elements with the identified class:

```
const images = document.querySelectorAll('.FFVAD');
```

3. Extract URLs

- Create an empty array to store the image URLs:

```
let imageUrlArray = [];
```

- Use the `forEach` method to loop through the `images` NodeList:

```
images.forEach(img => imageUrlArray.push(img.src));
```

- This code snippet iterates over each image element and extracts its `src` attribute (which contains the image URL) using `img.src`, then adds it to the `imageUrlArray` using `push()`.

4. Copy the Array

- To copy the array in a clean format, use the `copy()` function (this might be specific to your browser's developer tools):

```
copy(imageUrlArray);
```

Explanation

This script effectively automates the process of extracting image URLs from an Instagram profile page. It leverages the browser's DOM (Document Object Model) to identify and select the image elements, then extracts the URLs and stores them in an array for easy access and copying.

Benefits

- **Automation:** Saves significant time and effort compared to manually copying each image URL.
- **Efficiency:** Allows for quick and easy retrieval of image URLs for various purposes (e.g., backups, analysis, etc.).
- **Practical Application:** Demonstrates the power of JavaScript for automating web-related tasks.

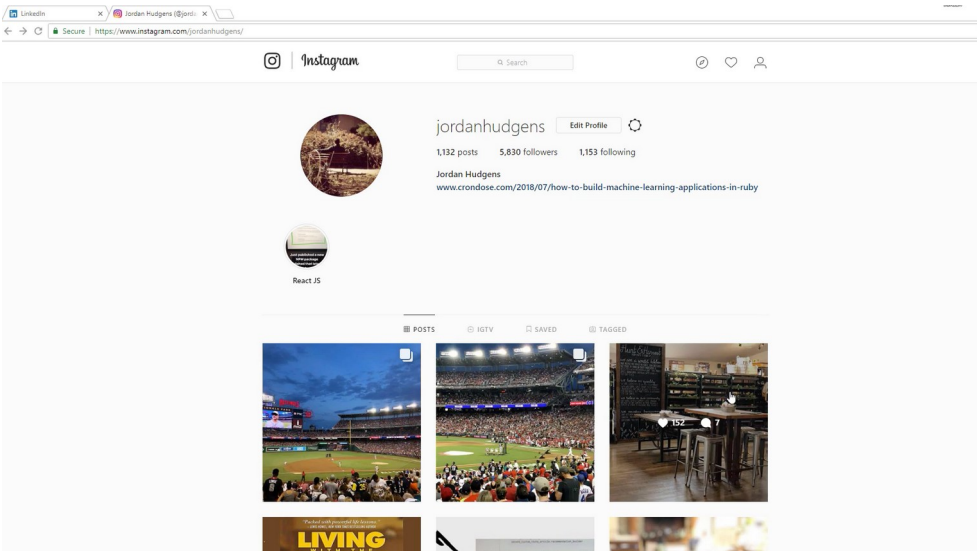
Important Notes

- **Instagram's DOM:** Instagram's page structure might change, which could affect the selectors used in this script. Always inspect the page to ensure you're targeting the correct elements.
- **Browser Compatibility:** The `copy()` function might not be available in all browser developer tools.

VIDEO:

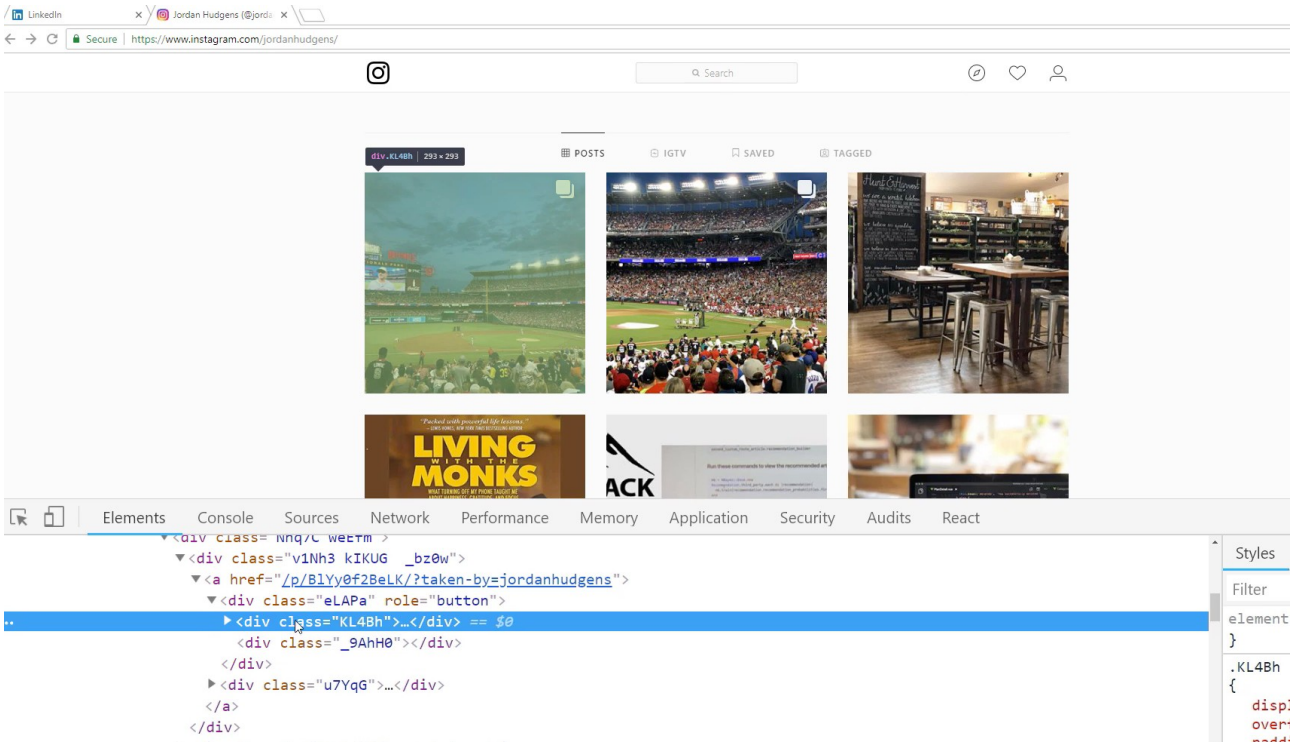
As you can see right here, I'm on my Instagram profile. You can be on yours if you want to try it out on yours. Some of the classNames may be different depending on when you're watching this. I have seen Instagram has changed a few of their structural things on their Website a few times.

I'm going to walk through the process on how I find everything, select it so that even if the `classNames` have changed you should still be able to accomplish the same goal. I want to grab each one of these images.

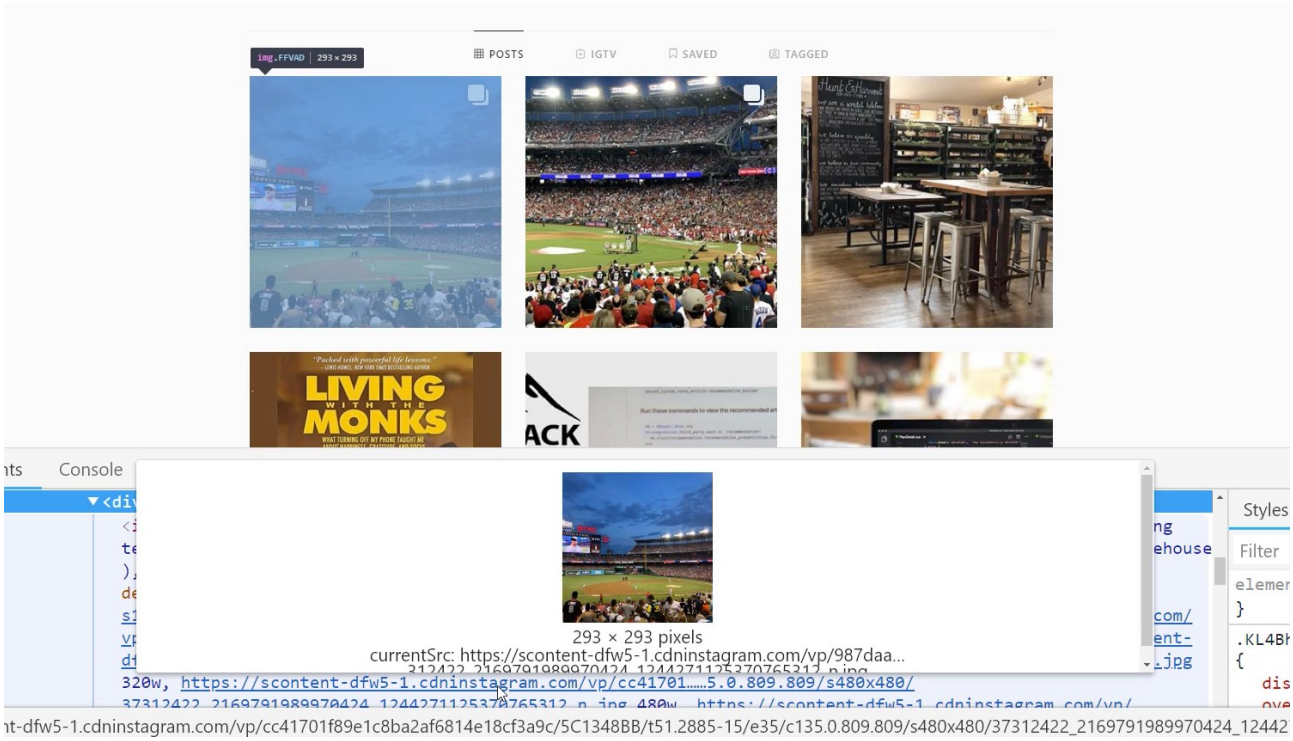


Now, if you click on the image, usually if you go to a website that stores their images in the traditional way, right here you'd be able to do something like say `copy image` or `save image as`. Instagram is a little bit trickier. trickier. You can't really do it that way. So we're going to have to use JavaScript in order to get all of these URLs.

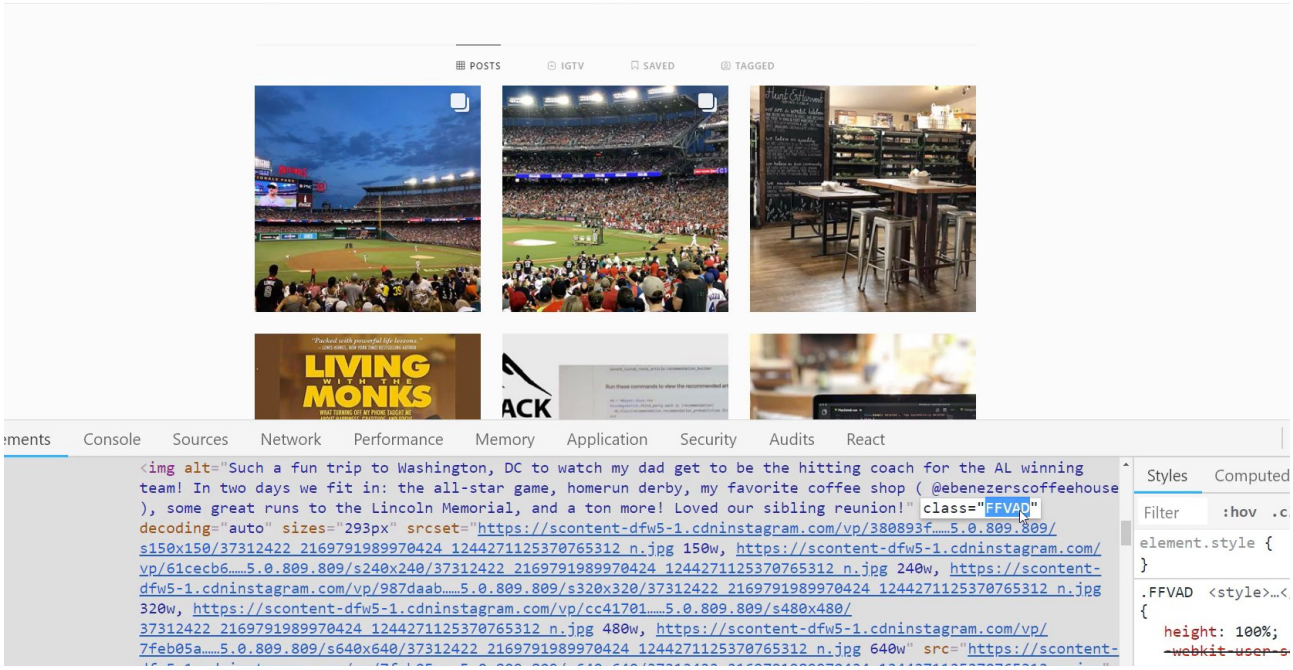
Let's start by right-clicking and then clicking on `inspect`. If I do that, what that's going to allow me to do is to see all of the classNames, and I want to go right here. You see where it says `div class` and then it has that `KL4BH`.



That's not what we want. That is the `wrapper DIV`, but if I extend that it's going to show all of the details. It's going to show the alt text and all of the images. Notice when I hover over each one of these images is showing each one these image links. It's showing me the image itself.



Now that is one way of getting the image, but that's still really manual. The one thing that this is telling us is: we now have access. If you come over here, you can see the className of the image it's that `FFVAD` class. I'm going to copy that.



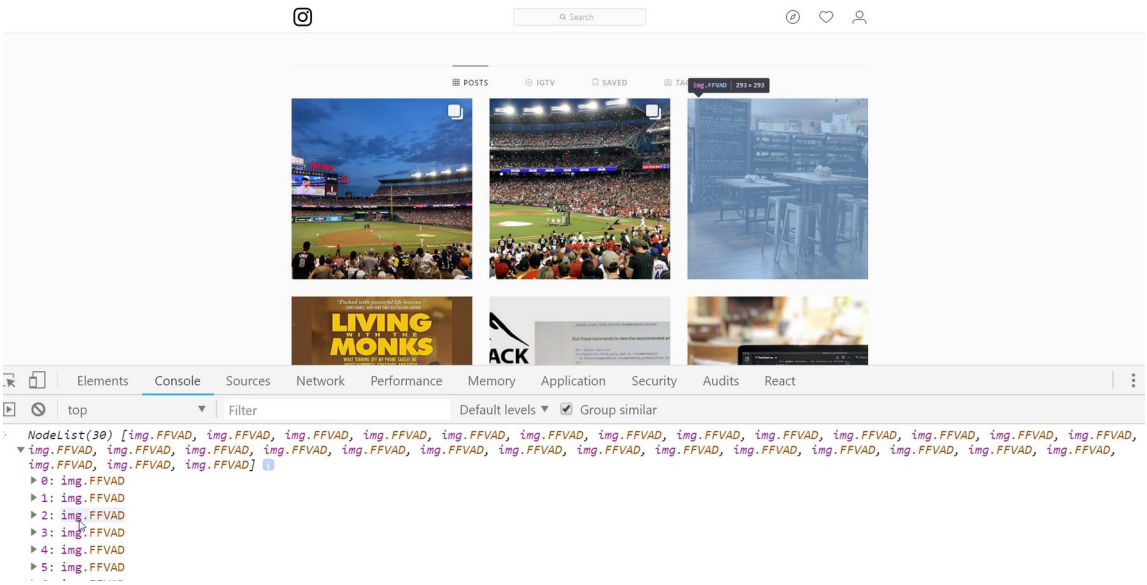
Coming over here, this time in the last guide we use the `dollar selector syntax`. Now I'm going to use just pure Javascript, so I'm going to say:

```
const images
```

Let's just get all of the images, and in this case whenever you're building the scripts you could say `let`, you could even use `var` for what we're doing here. It doesn't matter. I'm going to use `const images` and then:

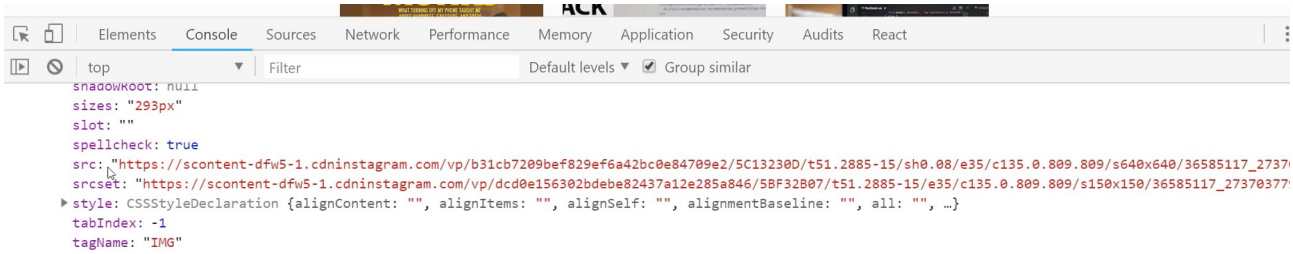
```
const images = document.querySelectorAll('.FFVAD')
```

Now if I run this, this is going to return all of the images in a `nodeList`. You can see if you extend this out, that it gives us each one of these. Looks like there are 30 of them, and if you want to go and hover over each one of them, see how it selects it on the page and it highlights that.



So if I want to go with that second image here, I can extend that. It gives me access to the `alt text` and all different kinds of elements. What I really want though is the URL. Let's scroll down.

Let's see exactly what we need in order to get access to the URL. Here it is. Right here you can see that we have this source, SRC, so that's what we're wanting.



Now what we need is we need to traverse through each one of these. We're going to loop through all of them, and then have this value returned. So let's come all the way down to the bottom, and then I'm going to type `control + l` and then `up` again just so we have access to our images.

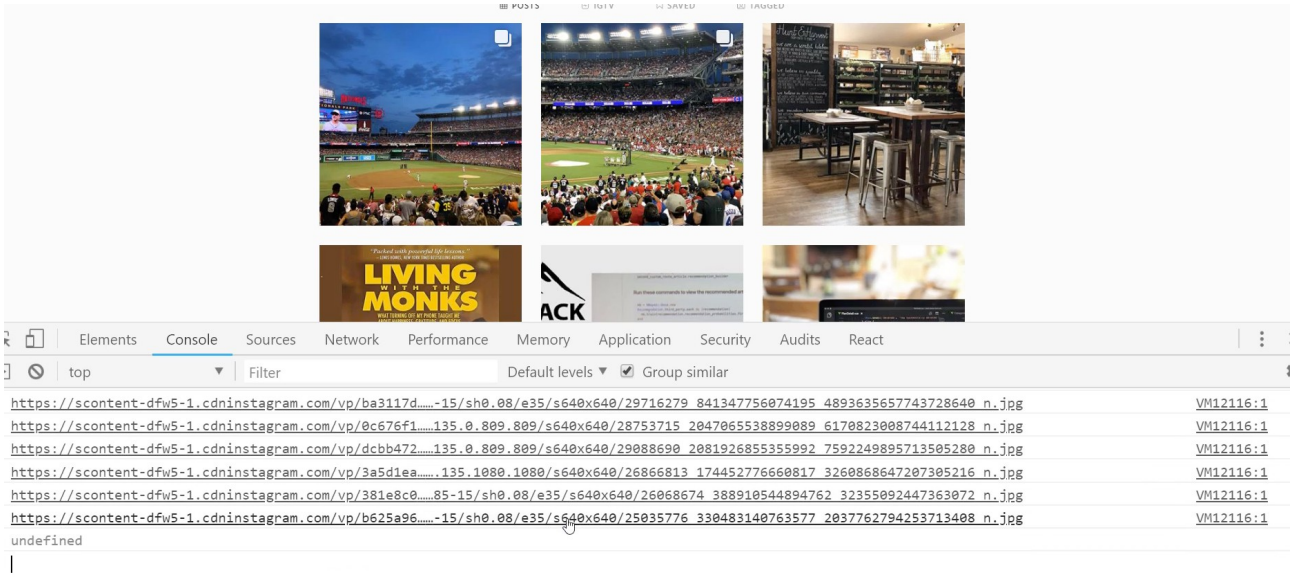
Now let's see if we can first just access the value. I'm going to say:

```
images.forEach()
```

This is going to give us the ability to loop over that node list and I'll just pass in a loop variable image. You could call this anything you want could call it `i` or `x`. It doesn't matter, but it's an image, so I'm just going to say:

```
images.forEach(img => console.log(img.src));
```

So `forEach` takes a function as its argument, so we're going to pass this in. For right now I just want a console log these values out, just so I know what I have access to. Let's see if this works. So if I run that, you can see that worked perfectly. It gives us each one of these image values.



If I click on this, you can see that there is one image. There is another one. Each one of those returns the image that we're looking for. So that is really cool. Technically, you could just copy and

paste all of this, but we can do better than that. If you try and copy this, notice how it also brings in the little `debugger line`.

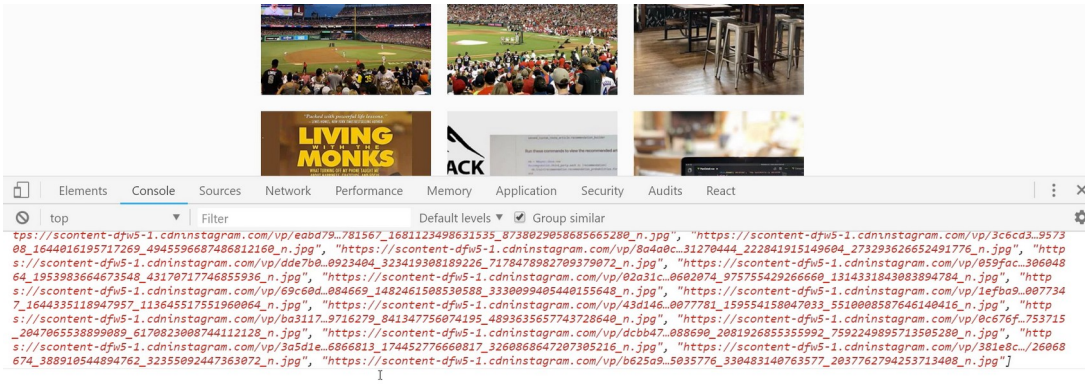
We can do better than that. So now that we know how we can access it, let's create an empty array. So I'm going to say:

```
let imageUrlArray = [];
```

Now if you just scroll up, or hit the `up arrow`, instead of `console.logging` that value what I want to do is I want to add to that the images array. If you remember how to do that, we can just say:

```
images.forEach(img => imageUrlArray.push(img.src));
```

Now what that means is the `imageUrlArray` is going to be filled up with all of those URLs. So if I hit enter now, it returns undefined, but if I call `imageUrlArray` now it gives us all of those images.



That is much better and gives us in that nice array format. Now you could say: "Okay, I can copy all of this, and then I could put it in a file or whatever", and kind of, but let's see what happens if we do that.

See how if I copy this and then we go into visual studio code. If I paste this in, see how it put it all on one line. That is not exactly ideal. I'm just going to delete that. I'm going to show you a cool little shortcut since we're on the topic of being able to automate our tasks.

If you type copy and then the item that you want to copy: this is going to put it on the clipboard in a much better format. So I'm going to say: `copy(imageUrlArray)`. Now if I come here, you can see it has taken all of the images off of the page, and it put it in a nice array format.

You could put this right into your own JavaScript code. You could use it if you're building test data, or if you're wanting to create a script that puts this in a database. Anything like that. Now you have access to all of it, and each one of these images are what you had on the page.

As you can see, if you had to do this, especially if you had to do this on a regular basis, then that would really not be very fun. Oh, that didn't let me copy that one line. Let me do that just so you can see that each one of these images is valid. Let me just copy that. There you go.

You can see that that is working perfectly. Now if you had to do this, and you had to manually go in inspect the element and grab all 30 of these, especially if you had to do on a daily basis, or if this is what you were one of the people working with you, needs to do this, it would really not be fun.

The way that I discovered how to do this was actually because I had a friend who is a social media manager, and they were having to do that exact task. They were having to come into profiles on Instagram. They would have to click on an image then perform some type of screenshot, where they grabbed it, and then they saved it somewhere. That kind of thing.

They had to do that on profiles all day long. So I created this script to show them how they could run one single script and grab all of the images, all at one time, and they were absolutely ecstatic. It literally took about 90% of what they did in regards to Instagram down on a daily basis, and they were able to work on other tasks.

That's one of the things that gave me the inspiration for building this entire section out. It was because I saw how if you know the tools that you have at your disposal then you can build some pretty cool things, even if you're not building programs.

Even if you just want to automate processes, then this is something that can be a really helpful skill to learn. You now know how to use JavaScript to traverse Instagram and grab the images from it.

Coding Exercise

Select all the below tags using a query

```
<p class="grab-these"></p>
<p class="grab-these"></p>
<p class="grab-these"></p>
<p class="grab-these"></p>
const selector = write-Your-Code-Here;
```

6.4 Auto Following Accounts and Hashtags on Social Networks

THEORY:

1. Inspect the Page

- Navigate to the LinkedIn "My Network" page or a hashtag page.
- Right-click on the "Follow" button and select "Inspect" to open the developer tools.
- Identify the button element in the HTML. Note its `class` attribute, which you'll use to target it with JavaScript.

2. Select the Buttons

- Use `document.querySelectorAll()`

```
let hashtagBtns = document.querySelectorAll('.mn-discovery-hashtag-card__action-btn');
```

3. Automate Following

- Use the `forEach` method to loop through the `hashtagBtns` NodeList and simulate a click on each button:

```
hashtagBtns.forEach(btn => btn.click());
```

4. Automate Unfollowing (when needed)

- Inspect the "Following" button to find its class attribute.
- Select the "Following" buttons using `document.querySelectorAll()`:

```
const followingBtns = document.querySelectorAll('.follows-recommendation-card__follow-btn');
```

- Loop through the buttons and simulate clicks to unfollow:

```
followingBtns.forEach(followbtn => followbtn.click());
```

Explanation

This script automates the process of following or unfollowing accounts or hashtags on LinkedIn by programmatically clicking the respective buttons. It leverages the browser's DOM to identify and select the buttons, then uses JavaScript's `click()` method to simulate user interaction.

Important notes:

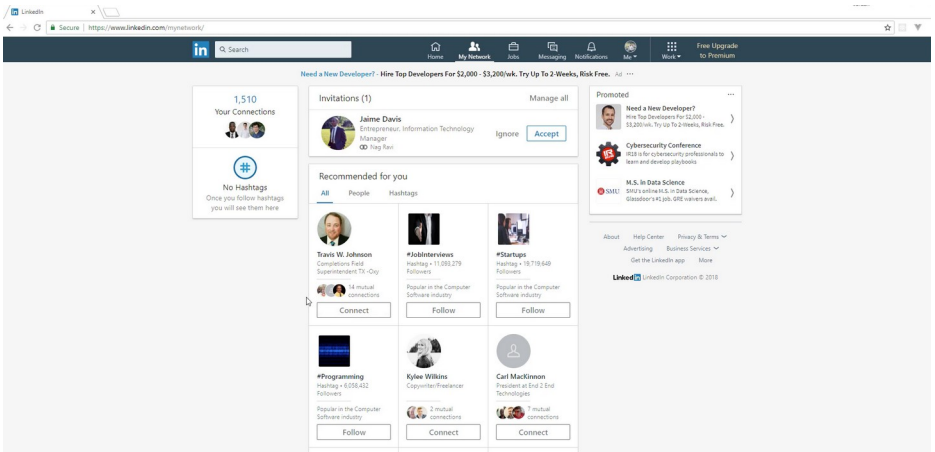
- **DOM Manipulation:** The script interacts with the Document Object Model (DOM) to access and manipulate elements on the page.
- **Selectors:** CSS selectors (like `.mn-discovery-hashtag-card__action-btn`) are used to target specific elements.
- **Iteration:** The `forEach` method allows you to loop through a collection of elements (NodeList).
- **Automation:** The script automates repetitive tasks, saving time and effort.

- **LinkedIn's DOM:** LinkedIn's page structure can change, potentially requiring you to update the selectors in your code.
- **Ethical Considerations:** Use this script responsibly and avoid excessive automation that could violate LinkedIn's terms of service.

VIDEO:

Now let's walk through how we can automate tasks on LinkedIn. Now, this is a very popular way of using JavaScript. If you're wanting to build out your network then this is a great way to do it in a very quick fashion.

Right here you can see I've gone to My Network on LinkedIn, and they give you all of these recommended items. Now you could also do this on a group page, and I'm going to use the hashtag page for this.



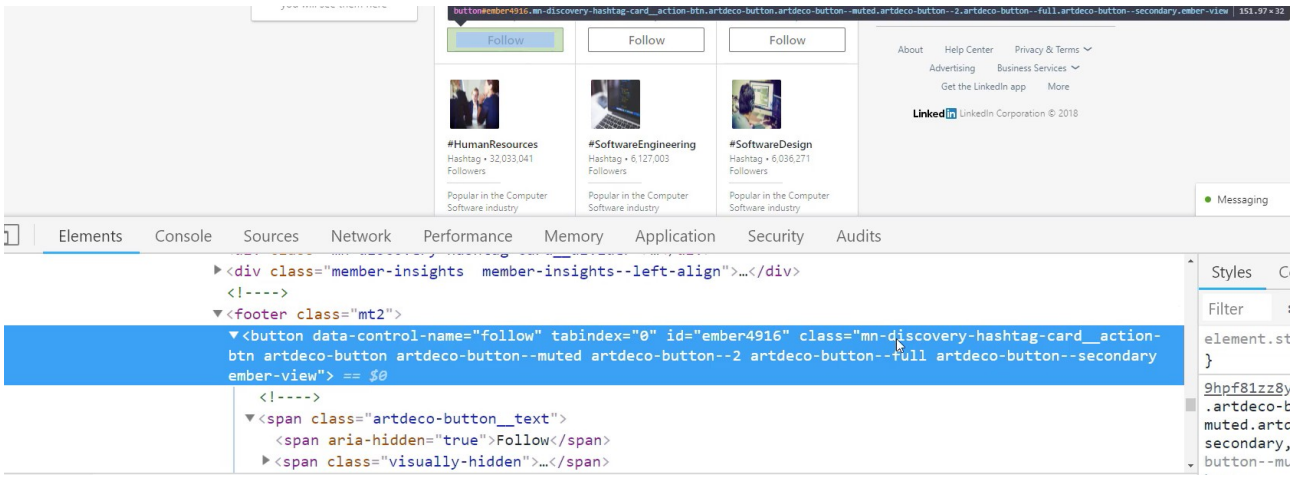
The reason why I'm going to be doing that is that if I did it for all: it would send it to a bunch of people, and I am not 100% sure if I want to add all of these recommended people. However, I know there are a lot of people out there and developers out there, who are trying to build out their network, and they simply want to run through and add a bunch of connections.

That's perfectly fine, but I'm going to show you how to follow a bunch of hashtags. The process is completely identical. I knew that because I ran a small test before filming this, and it did send out the **connect messages** and **connect invitations** out to everybody.

So this will work for people, for all, or for hashtags. I'm going to switch over to hashtags, and we can walk through exactly how this works. So right-click on follow and `inspect`.

Now, this is going to give me the actual text, because that's what I selected, but we need to move a little bit up further in the `DOM`.

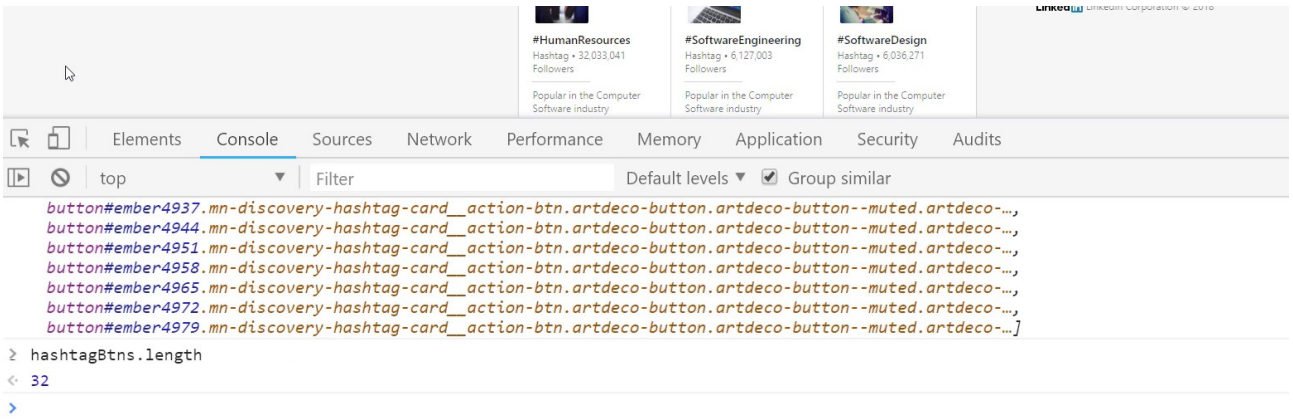
Right here you can see that we have the button. This button element is actually what we're looking for. So here in the button, this is going to be relatively straightforward, because we have a class that we can grab.



All I have to do is grab this `mn-discovery-hashtag-card__action-btn`. That's a nice long name. Hit copy, and then come to the console. Let me clear out any errors that we had. Right here let's run a test, and make sure that we have access to everything that we think we have access to. So I'm going to say:

```
let hashtagBtns = document.querySelectorAll('.mn-discovery-hashtag-card__action-btn')
```

Now what I can do is just run the selector, and now if I look for a `hashtagBtns` you can see it brings up all of the specific follow buttons that we have right here. If I wanted to check out the length. I can see I have `32` of those.



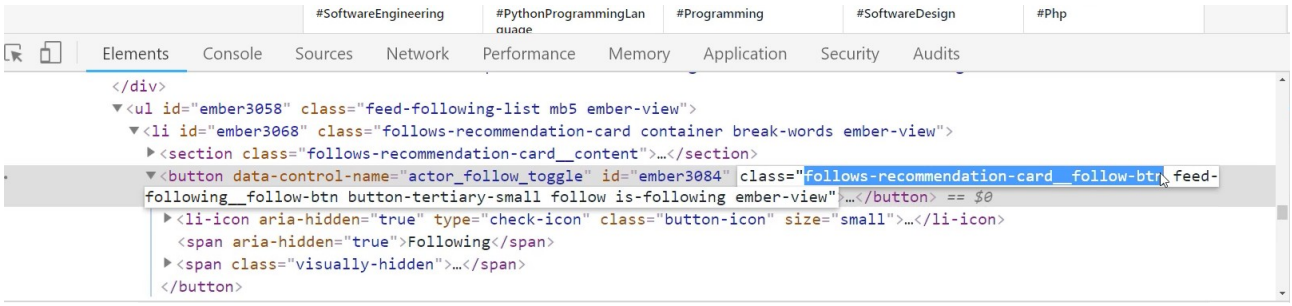
Now what we can do is we can just automate the process of following all of those hashtags. So here, I can say: `hashtagBtns.forEach()`. For each in JavaScript takes a function. So I'm just going to say:

```
hashtagBtns.forEach(btn => btn.click())
```

This is the `click function` that is available in JavaScript. Now if I run that you can see that it went and it followed every one of those hashtags. So now if I hit refresh on this page, you can see that I'm now following each one of the hashtags. So every one of those that was on there, I am now following.

Now you could do the exact same thing here, and I'm going to. I want to show you both sides of it, so I showed you how you could do it for **following**.

Now if you want to unfollow say all of these different elements. Let's see if we can do that. So I'm going to click `inspect` here, and so now you have this `actor_follow_toggle`. This is a control name. Then the `class`, you have `follows-recommendation-card__follow-btn`. This is what we're looking for, this class right here.



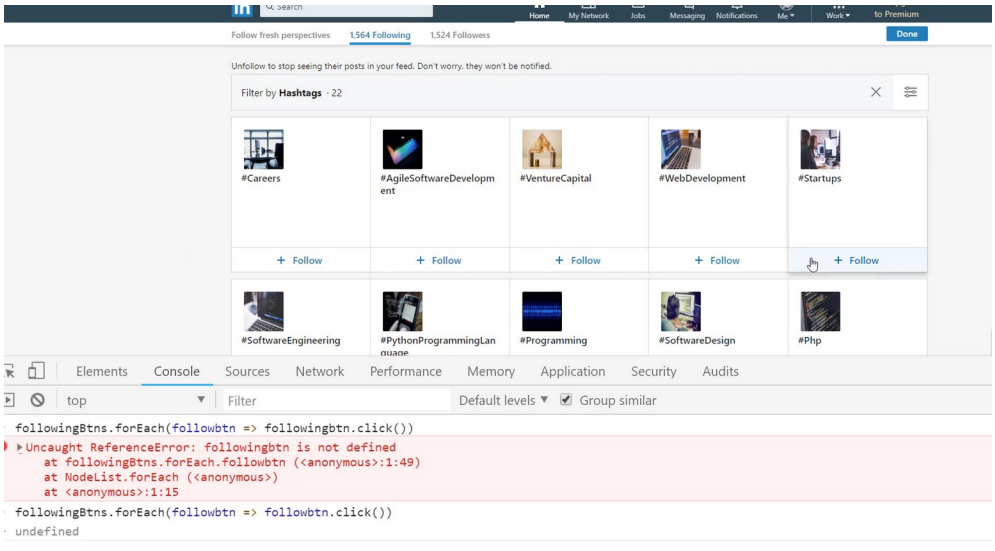
I'm going to copy this, and we're going to do the same thing. So I'm going to say, this time it doesn't matter if you use `const`, `let`, or `var` for the variable. I'm not going to change it, so I'm going to say:

```
const followingBtns = document.querySelectorAll('.follows-recommendation-card__follow-btn')
```

Run that, and now if I check to see all of those following buttons, you can see them right there. Now it can run the exact same process, so I can say:

```
followingBtns.forEach(followbtn => followbtn.click())
```

Now you can see it went in it unfollowed each one of those hashtags.



So with just writing a single line of code, we're able to follow all of those hashtags on the page, and then writing another very similar line of code we're able to go in and unfollow those. If that is something that you're looking to do, where you're trying to build your network or

in that specific to LinkedIn, but these processes that just walk through this could be applied to any kind of page.

You could do this for Instagram, you could do it for Facebook, you could do it for anything that you want to automate. The thing that gave me this idea was the Bottega Code School CEO asked if I could help him build a script because he was trying to automate the process of going and following a bunch of people in some of these LinkedIn groups.

So that's what gave me the idea to do that because I assume that if he asked for that, other people were looking to automate the same process. Now you know how to, and as you can see, it's relatively straightforward.

Remember, any type of script like this has two main steps. It first has the `query step`. That is where you go and you find what the class name is you're looking for, and then you query it and you store that value in a variable.

Then from there, the second step is to `perform the process`. In this case, the process was simply clicking on the button. The process could be anything else. It could involve other steps if you need to say fill out form elements and then click or whatever it is that you need to do.

As long as you have that element selected when you have `forEach` and you have the method like this what you're able to do is treat each element as if it was you who is going and you were clicking or you were filling out a form or whatever process you're looking to do. You can simply have the code do it.

That's what this is all about, is being able to automate the entire process using JavaScript. Great job. You now know how to fully automate any kind of process that you need to do inside of a browser.

Coding Exercise

6 Snakes have made themselves at home in your boot. Use the query selector all to locate all of the snakes!

```
<div id="boot">
  <div class="snake"></div>
  <div class="snake"></div>
  <div class="snake"></div>
  <div class="snake"></div>
  <div class="snake"></div>
  <div class="snake"></div>
</div>
```

const nodeList = write-Your-Code-Here-to-select-the-snakes;