Week 1: Introduction to JavaScript ES6

Video Transcript

Video 1 – Introduction to ES6

Given the requirements of the course, no doubt you have some experience with JavaScript. However, we will be using quite a bit of ES6. The new patterns, the new ways of expressing intent, the new shortcuts that have been built into ES6, and because of it, we wanted to give you a chance to look at those patterns on their own as we set out in this course. We will be looking at default parameters and how we can simplify those calls and express our intent in a better way across APIs, we will be looking at template literals that have improved dramatically the way in which you can use strings.

In fact, it turns out to look a lot like the templating languages that have existed in the past in the front-end. We will also be looking at spread and the structure, especially around modification of lists. And you'll see how we can do that in a more compact way. If you have not used arrow functions before, this is a good opportunity, once again, to be able to leverage this more compact syntax. You'll see that many times when you are using callbacks, it leads to more readable code, which is always a good thing. We will also look at the patterns that are present when it comes to asynchronous computation.

And we will see how async and await provide a simpler-to-author and read syntax when it comes to expressing asynchronous computation. We'll also look briefly at modules and how to wrap some of your syntax and how you can take advantage of some of the encapsulation that takes place when you use modules. Finally, we will look at the differences between let and const as opposed to the vars that we have used in the past. And some of what we can now do using this new type of encapsulation, especially when it comes to the scope of variables. So, with that in mind, go ahead and dive into ES6 and take a look at some of what you can leverage later on in the course.

Video 2 – Default Parameters

Let's look at parameters and functions by writing a simple example, we're going to write a function and we're going to multiply two numbers. We will call it 'multiply' and the parameters that we will take will be '(a,b)', and the body of the function, we're simply going to 'return a*b;', and now we're going to do a couple of console logs to call this function. We're going to enter 'console.log'. And we're going to call the function '(multiply'. And we're going to give it a couple of parameters, we'll enter '(5,4));'. Now, if we reload this in the browser, I have here the file that I'm working with that I've called params.html.

You'll see there that as expected, we get 20. Now, if we call the function again, but leave out the other parameter, we then get the following. We get not a number because that value does not exist. And when that multiplication is made, not a number is returned. So, we could write defensively to avoid this case. And we could do something like 'if (!b)' does not exist and we could check for it that way. We can then assign 'b' the value of '1;', a default value. And then if we reloaded our code, we'd see here that we now get 5 for the case where the parameter is not provided, it's missing. Now, this works.

However, we can do this in a cleaner way by using default parameters in ES6. And the way we would denote it is the following. This is equivalent to what we just wrote and if we reload the page, you can see there that we still get 5 for the second console log. And simply to be more explicit, we could enter 'undefined' for the second parameter. And if we reload the page, we once again get 5. So, the full parameters are something that you could write defensively to address. Or you can simply use ES6 syntax and provide the default value within the parameters in the function signature, as you can see here directly in the parameters.

Video 3 – Template Literals - Multiline Strings

Say you wanted to put the four lines of Humpty Dumpty into a variable. A common way to do this is to use string concatenation. So, let's go ahead and show that. We're going to go ahead and create a variable called 'message' and into that, we're going to put in these four strengths. So, I'm going to go ahead and copy that into here, and then I have to enclose in quotes, each one of these strings. Let me go ahead and do that first, and then we will add the plusses and note that I have to handle the apostrophe that is in here. I'm going to go ahead and put in ' \ '. Then I'm going to have to go ahead and add a '+' to each of those lines. And this one doesn't need it because it's the last one.

So, you can see there that works. It's a bit messy, but we can do it. Let's go ahead and write a 'console.log()' to confirm that we did that correctly. Let's go ahead and reload the page and as you can see there we get Humpty Dumpty. Now, note that we have it all in one line and we could further work on this to put inline returns. But let's go ahead and look at a different way to do this. And we can do this using ES6 string's backticks, which allows us to define a block of text without having to do string concatenation. So, I'm going to go ahead and delete what we have here and I'm going to go ahead and copy this once again. However, now I am going to denote it using the backtick And I'm going to go ahead and reload the page.

And as you can see there, we have it once again. Now, note that we have the exact same thing that we have on the page. That is if we wanted to get rid of that space at the beginning of each of those sentences. You can see there that we can do so by removing the space within the code. So, as you can see there, that is much simpler, it removes a great deal of work. It removes a great deal of mistakes that are often made. And so, this is an easy way to create or to put a block of text into a variable. Now, in case you're wondering what you would do if you wanted to put it all in one line. We can put in a ' \ ' for that. And I'm going to go ahead and show it here. And if we reload the page, we get it all in one line. So, this is a much friendlier, as mentioned, way to work with strings, and is one that we recommend you use.

Video 4 – Template Literals

Say you wanted to nicely output a date and you had the following values. As you can see here, you had 'thursday';, 'december'; '15', '2050';. And you were using a function called 'prettyDate' that passed those parameters. So now, we're going to go ahead and implement that function. And one way you could do this would be to use string concatenation. We could first create the 'message' that we're going to return. And then we're going to start off by putting a friendly message. We will say 'the date is '. Then we will '+' to this string the parameters. The first one will be 'day'. We will then '+' a ' ', ' '. Then we will '+' to that the 'month' to that we will '+' a space now, no comma just a space. Then to that, we will add the 'dayNumber'.

All of these parameters again were passed in. And we are going to now use a comma, in this case, after the number. Then we will '+' to that the 'year;', and then we will be done. Now, we just need to 'return' our 'message;'. And this should write out to the console. If we go ahead and reload the page, we can see there that the message we get is the date is thursday, december 15, 2050. This certainly works. However, there's a much cleaner way to write this using ES6 template literals. And so, I'm going to go ahead and delete this part here. And we're going to use a template literal with placeholders for expressions. And so, I'm going to write a sentence that has everything we need as far as text. And then I will add placeholders for the variables. So, the very first one I'm going to enter is '${day}', as you can see there.

And I'm going to grab just this part so I don't have to type it in every time. Then we're going to write the '${month}'. Then we're going to write a space, and the '${dayNumber}'. Then we're going to enter a comma. And finally, we're going to enter the '${year}';'. And as you can see there, I've just toggled the line wrap. So, this is a much more intuitive way to write that line. And let's go ahead and see if we get the same thing. And we do not. And the reason for that is that we're not using backticks. So, let me go ahead and do that. And you can see there that the editor itself recognizes that we're now using placeholder expressions. And if I reload the page, you can see there that the date is thursday, december 15, 2050. Once again, this is a much more intuitive way to work with strings, and it allows us to ride very clean templates.

Video 5 – Embedding Expressions

You can also embed expressions inside of your template literals. Let's do a small example. Let's 'let a = 5;'. Let's 'let b = 10;'. And then let's construct a string first using concatenation. We'll enter a 'console.log()'. So, we can look at this string. Inside of it, we will enter the following message. '('a+b is: ' ' is inside of the string, we will evaluate 'a+b'. So, we will enter '('a+b')'. And then we will add to this string a small message simply to extend this string. And I will say, ' ' and not ' ' here, I will multiply instead of adding, so '(a*b));' So, let's take a look and see what that evaluates to that and as you can see there we get a+b is: 15 and not 50. So, let's go ahead and copy that line. Then let's use a template literal. And in this case, we will use a backtick. And so, we will start it there and we will end it here.

Let me go ahead and just go to this location. And now we no longer need concatenation. And so, we will simply enter the placeholder. Inside of the placeholder, we will embed our expression. So, in this case, I'm going to enter then curly braces, and inside of that I will enter '${a+b}'. And in the second case, I will enter '${a*b}` '. So, let's go ahead and evaluate this expression, but let me go ahead and put in that closing parentheses. And it looks like I did not enter the right value there. And you can see there that we get the same. However, the template literal is shorter, it is also cleaner. We don't have to do all of this editions. The mixing of code and, or embedded expressions and string is also a lot cleaner. And so, this, as you can see, makes it easier to work when you are building strings. Then if you're using concatenation.

Video 6 – Destructuring Assignment

The structuring assignment makes it possible to impact values from arrays and objects into variables. Let's start off with an array here, and I'm going to create a couple of variables. I'll create a variable 'a', then one called 'b;' And then I'm going to create an array that holds two values, '[10, 20];'. And we're going to destructure assignment into two variables as well. And so, we will receive into '[a, b]' as follows. Now, we're going to write the values to the 'console' to see that we in fact have the value that we think we have. The first one will be '(a);', the second one will be '(b);'. And let's take a look at that.

Reload the page here. And as expected, we get 10, and 20. So, we have destructured that array into a and b. Now, let's go ahead and add a few more parameters, '30, 40, 50];' into the array. And let's add one more variable and we will call it 'rest;'. And this is appropriately named because it allows us to get the rest of the parameters, in this case in the array. And we can denote that using the following syntax by entering three dots plus the variable name that you're going to use. So, let's go ahead now and write to the 'console' the value of '(rest);'. And let's go ahead and reload the page.

And as you can see there we get the rest of that array. Destructuring turns out to be a very popular pattern within the language. And is one you will see in many places where modern JavaScript is being used. JavaScript that uses ES6. So, this is one that you want to practice. You want to play around with, make sure you get some comfort before you move forward. Now, before we close, let's take a look at an example with objects. You will see that it works very much the same way that arrays do. And I'm going to simply replace that line.

I have the same values here, except that I'm not going to 50 I'm stopping and 40. You can see there that we are destructuring into a, b and rest. And so, let's go ahead and reload the page and let's see if that works the same way. And you can see there that it does. We have 10 again, we have 20 again, and then we have the properties of c and d that are captured in rest. So, once again, this turns out to be a very useful pattern, it helps you to clean out your code. It'll help you understand a lot of the modern JavaScript that's being written and this is one that you want to practice.

Video 7 – Destructuring Assignment - Functions

You can also use destructuring with return parameters from a function. Let's go ahead and write a 'function' here that will take no parameters and we'll simply 'return' an array of the following values, '[1,2,3,4];'. And we're going to call it and use destructuring to take the first two values of that return array or that array that the function is going to return. So, I'm going to go ahead and call the function, and then we will write to the 'console' to verify those two values. We're going to do it here for '(a);' and then '(b);'. Let's go ahead and reload this page and as you can see there we get 1, and 2 as expected. We call the function the array of [1,2,3,4]; comes back.

We destructured that into two variables called a and b and then we write those to the console. We can do the same when we call the function, and the function can control which parameter, say, of an object which properties it wants to receive. So, let me go ahead and paste an object that I've written beforehand here. and I'm going to remove the code that I had typed. You can see there that I have an object called 'user' that has an 'id:', 'firstName:', 'lastName:', and an 'age:'. And then I'm going to write a function that is going to take as a parameter that object and I'm going to call it 'userId()'.

Then in the parameters, I'm going to specify which of those properties I actually want to receive and I'm going to pass in a moment the whole object. And here I'm simply going to 'return' the 'id;' to see that in fact, we are receiving only that. And we're going to call it from the 'console', passing the object that we've written. So, I'm going to call '(userId' with '(user));' and we're going to see what that returns. So, let's go ahead and reload it and as you can see there, we get the id. So, we wrote to review, we wrote a user object.

We then wrote a function called userId that destructures the object that is being sent to it and only takes id, the first property, as you can see there, and then returns that property. And then we call that function, and we pass in the user object, and then we write to the console what that function returns, which is 209. To verify here, we could enter '(age));', changed this to '({age})', changed this to 'age;' as well. We can go ahead and reload the page and you probably caught the error that I passed age instead of the user object, I have to pass the whole object, which then gets destructured, and only age is taken and when we reload the page, you can see there that we get 19 as expected.

Video 8 – Spread and Rest

Let's look at an example of rest and spread with a list of numbers. We'll start off with '// REST'. We'll create our list. We will call it 'numbers' and the array will simply hold the numbers '[1,2,3,4,5];'. And then we will use the REST notation to be able to extract the first position and then the rest of the numbers in that array. And we will do that in the following way. This will be '[first,' and then the '...rest]' and we will get that from our list of 'numbers;'. Then we will write to the 'console.log(first);', the first position. And then we will write the '(rest);' of that list. I'm going to go ahead and reload the page.

And as you can see there, we get the first position and then we get the rest of the numbers, which is [2, 3, 4, 5]. So, note that we use REST to be able to collect the rest of the items in that array into a variable of our choosing. Next, let's take a look at '// SPREAD' which is the opposite. Here we will unpack the items in the array into variables. So, in this case, I will create, let me give myself some space here. I will create an array called '[a,b,c,d,e]'.

And then I will use 'numbers;' once again. I will then write to the 'console' to check our spread. And I will write the values here. We'll enter '(b);' '(c);' '(d);' and '(e);' And now, if we reload our page, you can see there that we get 1 2 3 4 5. So, while the first one collects the values, the rest of the values into one variable. When we use SPREAD, we unpack the items into a number of variables, as you can see here. In this case, we went a through e. And in the example with REST, we simply put it into a variable called 'rest'.

Video 9 – Spread - Value and Reference

Let's say we have a list of numbers, an array of 'numbers', and the values we have are the following. Now, let's say as well that we want to preserve this list, but we want to change the third position. One thing we could do was to make a copy and let's call that 'mycopy' and assign to that 'numbers;'. Now, we could take 'mycopy', change the third position. Let's assign it here, the value of '9;'. And then write to the 'console'. First '(mycopy);' to confirm that we have made the change. And then the original list, which is '(numbers);'. And now let's go ahead and reload the page in the browser.

Now, as you can see there, both lists have been changed. They both have for the third position, the number of 9. And that's not what we wanted. And the reason for that is that when we made the assignment on line 4, we made a reference to numbers;. We did not create a new copy of that array. Now, we could do that by iterating through all of its elements and creating any array. But there's a simpler way to do that. And we can use that with the spread notation within JavaScript.

What we will do is we will create a new array. And then we will use to spread notation to be able to get all of those elements. And so, if we now reload the page, you can see there that my copy, the first console.log, has changed the third position and is now 9. And you can see here, however, the second array, the numbers array, has not been changed. And the reason for that is that we no longer have a reference. Now, we have a new copy that we can modify without altering the original list.

Video 10 – Spread - Merge Arrays

As you can see here, we have two arrays, two list of numbers. The first one 'l1 = [1, 2, 3];'. The second one, 'l2 = [4, 5, 6];'. Let's consider that we wanted to emerge both of these arrays. One thing we might try is we might make a 'newList', including both of those in an array. Let's say we did 'l1' and 'l2' as follows. And then let's take a look at what happens in the 'console'. Let's go ahead and write it out in 'JSON', so we can see the values. I'll enter here, the '(newList)'. Let's go ahead and reload the page. And as you can see there, we have in fact brought both of those list in.

But they're still two independent arrays, not one array including all the numbers. Now, another thing we could try is we could put 'l1' here, and see if that work. Now, we would write out '(l2)'. And as you can see now, we don't have two arrays inside of another array, but we do have one array that is still inside of a larger array with 4-5-6 next to it. So, one way we could do this is we could use the spread properties, and we could put '...' in front of 'l1'. Now, we could reload the page.

And now, we get exactly what we need. Not only that, we could create a new one, let's say, 'newList'. And then inside of this one, we would enter here '[...l1,' then '...l2];'. And now, we no longer need it here. And so, let's go ahead and take a look on our '(newList)' and reload the page. And you can see there that we do get the same thing again. So, this is a very convenient way to merge two arrays by using the spread notation within JavaScript.

Video 11 – Enhanced Object Literals

As you can see, I have two variables here. The first one is 'a', which has been assigned the string of ' 'one';'. Then I have 'b', which has been assigned the string of ' 'two';' And next we're going to create an object literal. I'm going to go ahead and assign it to a variable called 'obj1'. Then I'm going to write my object literal I'm going to give the first property, the name of '{a:' and the value of 'a,'. Then the second property is going to be 'b:', and it's going to have the value of 'b};'. If we were to write to the 'console' one of those properties, let's say '(obj1.b);'. We would see the following. As you can see there, we get two as expected.

Within ES6, we have enhanced object literals notation that allows us to simplify this. We can simply create a new object as follows. I'm going to create this one 'obj2'. And instead of repeating a: and a, and b: and b, I can simply do the following. And if I were to do the 'console.log' once again. And in this case, write '(obj2.b);', And we reloaded the page. Let me just enter my line termination there. You can see that once again, we get two. So, this is the same result. However, we can simplify our notation by removing the name of the property and then the value of the property if we're going to use the same for both.

Video 12 – Arrow Functions

We can create a function to return a greeting, the following way. This function will 'return 'hello';', and we can confirm it by writing it to the 'console', and invoking that function. Let's go ahead and reload that page. And as you can see there, as expected, we get 'hello' back. Now, using arrow functions, we can do the same thing in a more compact notation. I'm going to call it 'greeting', once more. Then I'm going to use the '=>' notation. And then I'm going to repeat the same call. And if I reload the page, you can see there once again, I have gotten 'hello' back. Now, note that if we have one statement, the return is implied.

So, we no longer have return. Also, the curly braces are gone, as well as the need for the keyword function. Now, if we had much more in that function, we could do use curly braces just as before. We can use return as well. And if we had 'parameters' in a small example, we could do the following. Let's write a 'add' function. And in this case, we're going to take two parameters to add. Then our '=>' function notation. And we're going to simply return 'a+b;'. So, if we enter 'console.log' or if we write 'console.log', and then 'add', and give it a couple of parameters. And we reload the page, as expected, we see '2'.

Now, let's go ahead and take a look at an example with callbacks. I'm going to go ahead and delete the code we had, I'm going to paste in array of 'labels' for 'numbers'. We're going to ride a classic callback. We're going to catch our output in a variable called 'output'. Then I'm going to 'map' to the items of that array the following function. I'm going to call the item I'm passing in a '(label)'. Then I'm going to write the body of the function, and I'm simply going to 'return label. length;'. Now, once I have it, I'm going to write to the 'console' the '(output);'. So, let's go ahead and reload this page. And as you can see there as expected, we get the length of ' 'one' ', then the length of ' 'two' ', or the string length, and then that of ' 'three' '.

Now, let's do the same with an arrow function. I'm going to go ahead and copy-paste this here. And I'm simply going to write here 'arrow function'. Then I'm going to remove this part. Note that if we only have one parameters, we do not need the parentheses. My '=>', and then I'm going to remove everything until 'label.length;', and then everything else until the closing parenthesis. And then I'm going to write to the 'console'. Now, I'm to go ahead and reload. And as you can see there, we get the same, the same output as before.

So, as you can see here, we can use arrow functions to write more compact code, easier to read, as well. As you start to write arrow functions, a good recommendation is to first write it in the traditional function notation way where you are writing the function just as you have in the past and then replacing it with the '=>' notation. As time goes on, you will develop some intuition as to where you prefer, and it helps for clarity to have the function keyword and the classic definition, and where it really aids to the clarity to have a more compact arrow notation.

Video 13 – Async And Await

Promises are great. However, the .then notation and the chains derive from it can turn out to be pretty messy. To address this language has provided cleaner syntax and that is asynchronous functions, async functions, and the keyword await. As you can see here, async functions return a promise without having to create a promise inside of the function and await operator is used to wait for a promise. So, let's go ahead and take a look at an example. As you can see here, I have a very simple 'function' called 'resolveAfter2Seconds( )'. And inside of it, I have a 'Promise'. And inside of the Promise, I have a 'setTimeout( )' that will fire in two seconds. I'm now going to consume that with an 'async function'.

This is part of the keywords that I just mentioned. And I'm going to call the function 'asyncCall( )' because it's going to call the function with the Promise and inside of it, I'm simply going to write to the console the following. I'm going to call '('calling');'. And then I'm going to catch the result in a 'const' called 'result'. And I'm going to wait using the keyword 'await'. And then I'm going to call the function called 'resolveAfter2Seconds( )'. I'm then going to write to the console the '(result);'. Now the last thing I need to do is to invoke that function, the async function. And now let's go ahead and load that code in our browser. So, you can see it's calling, pausing, and then it is resolved.

So, the async function first writes to the console that is calling. It then waits for the result and the resolution of the Promise, which in this case resolves in two seconds. And then the message that is sent back I am writing to the console. Let me go ahead and reload that once again. You can see it's calling and now it's resolved. Now let's go ahead and take a look at an example that is more involved as you can see here, I have code very similar to one I've used before. I have three functions. Each of those functions has a Promise, and inside of that Promise, there is a timer. Now, in this case, note that each of them will execute and three seconds, and in this case, because we're using, or we will be using async and await, we will be executing in series, that is one after the other.

So, let's go ahead and write our code. I'm going to go ahead and scroll a little bit down. I'm going to create a 'const list' where I will hold my functions, it's going to be function '[a, b, c];'. And then below it, I'm going to write the 'for( )' loop where I will step through each of those '(fn of list)'. Then I'm going to write my for loop body, the 'const result' that will be coming back once I call the function. However, in this case, I'm going to use the keyword 'await'. And then I'm going to write my '(result);' to the console. Now because we're using the keyword await, we must use it inside of an async function and so I'm going to create a wrapper to do that. And I'm going to put the code that I've written inside of it. Let me clean up my code here a little.

And as you can see there, the little red underline is because I haven't given it a name. But in this case, we're simply using it as a wrapper. So, I'm going to wrap all of it in a parentheses and then execute it as soon as this code is reached. So, let's go ahead and take a look and see what this looks like in code in the browser. Let's go ahead and reload that page. And we are now waiting to see if we can see the code and you can see there a, b, then c. And note that in between each of those appearing on the screen, we get that three-second delay. First a, then next will come b, and then the last one will be c. So, as you can see, these are sequential instructions that are taking place. As opposed to the previous example where we were executing in parallel.

Video 14 – Modules in ES6

Writing code has a great deal of complexity. And because of it, there had been many mechanisms that have been built into languages to allow us to decouple and package a lot of the functionality. This has been particularly difficult within the browser environment due to the different languages. We have HTML. We have CSS. We have JavaScript and some of the limitations of the browser itself. Modules which are new to the language part of ES6, allow us to cleanly wrap a lot of this functionality in ways that were impossible before. Allowing us to keep our code cleaner and more maintainable. So, let's go ahead and write one of these modules.

I'm going to go ahead and start off by creating a function here for a 'user'. And it's going to take two parameters, a 'name,' and the 'age'. And then in the body, we're going to construct an object. And that object is going to take the 'name,' the 'age,'. And then we're going to have a couple of functions. The first one is going to be to 'printName:'. And I'm using the '=>' notation here. And I'm going to write to the console when this is invoked. And we're simply going to write out the name. Then I'm going to have one to 'printAge:'. No parameters again, '=>' notations to write the '(age)' to the 'console'. Then we're going to go ahead and return the object.

So, this is a function that composes objects. And now, comes the new part of the language that allows us to create modules. And the keyword that we're going to use here is export. And when we consume this from other JavaScript language files, we're going to import it. So, that's what we're going to do next. And before I do that, let me just point out here that I have three files. I have an index.html, then I have a main.js, and then the module that we're writing. The piece of functionality that we're packaging. And now, I'm going to move to main where we could potentially be consuming many of these package bits of functionality.

But in this case, we will simply 'import {user} from'. And then we need to give it the path that is ' './' ', pointing to the current directory, and then the file. Now, we're going to go ahead and use user. And we're going to create a 'student'. And 'user()' is going to take, and I'm going to put in here a couple of parameters. In this case, I'm going to enter '('peter parker')'. Then his age, which will be '18'. And then to test it, I'm simply going to call on those functions that 'printAge()'. Well, let me do the name first, 'printName()'. Then the one that prints the age. And now, we are ready to write to the console. Remember these functions write to the console. I'll show it here. And now, however, before we do this, we need to bring in that code into our index.html.

So, here we're going to enter the '< script >' tag, then the source. And then we're going to enter the name of the file, which, in this case, it's ' "main.js" '. Now, let's go ahead and run that code. A couple of comments to make here at the outset. One is that I am using a server. I'm using a small server that I install with npm. If you have not used npm before, you can wait till the lessons on npm, and then come back and try this part. Or you can install some other small server that you are familiar with. Now, let's go ahead here and reload the page. And as you can see there, I've gotten a error message. And the error message is due to the fact that I have not told the HTML part the script tag that I'm using a module.

And so, I can do that here by entering the 'type', then entering ' "module" '. Now, I'm going to go ahead and go back to my code and reload the page. And as you can see there as expected, we get peter parker and 18. And note that my HTML page is pretty clean. And nowhere within that page am I referencing any of these specific packages of code, in this case, user. Yet, I get the functionality as expected. Now, let's go ahead and add a ' "target" ' '< div >' here. To illustrate, my autocomplete is not working, and now it is. Now, let me go ahead and give it the identify it, identifier of ' "target" '. And we're going to use this '< div >' here to populate some content on the index.html page.

So, I'm going to go ahead here and get a handle first on that element. And I'm going to call it 'element'. I am going to access that element through, through the document. Then I'm going to use 'getElementByID'. I will enter the name of the id, which was target. And once we have that, then I am going to construct some HTML that we're going to add to that element once we construct it. And my HTML is going to be a template literal, literal. And inside of it, I'm going to say 'The student name is'. Then I'm going to reference or embed some of the data that we already have. And in this case, it's '(student.name.')'. And then I'm going to continue that string. I'm going to say, 'the age is'.

And I'm going to once again embed, an expression or statement here. And I'm going to enter '{student.age}' Let's go ahead and end that line. And now, we're going to set the 'innerHTML' of our element with the HTML that we just built. Now, I'm going to go back here to the page and I'm going to go ahead and reload it. And as you can see there, we are now writing to the front end to the name is Peter Parker and the age is 18. So, as you can see, you can use modules to more neatly package the computation that your writing and to provide code, that is clearer to read, that is more maintainable, that is cleaner.

Video 15 – Function Scope

When we create an assigned variables using 'var' as I'm going to do here, I'm going to create a variable of 'age', and I'm going to assign it the value of '20;'. And I'm going to write it out to the 'console' so I can keep track of its value. When we declare an assign using 'var', we can reassign. So, if I wanted to, I could change here the value to '30;' of 'age', and as mentioned I will keep track with 'console.log();' here. And we could even redefine. We can do all of this without any errors from the language. And so, if I were to reload the page here, you could see there that I have 20 originally, then 30, then 40.

Now, we could protect our variable by using a function, and that's what I'm going to do next. I'm going to create a 'function'. And I will call it 'userInfo()'. And inside of it, I will create a variable using 'var' called 'age'. I will assign it our original value, which was '20;', and I will 'return age;' And then I will write to the 'console'. Here, let me enter 'console.log()'. I will write the output of this function, which is the value of age. Then I will write to the 'console' once more the value of '(age);'.

And I'm going to comment this out for a moment. And if I where do reload the page, you can see there that we get 20 as expected.

If I reload the page now, you can see there that age is not defined, so it only exists inside of the function. This is what we call function scope. It is protected, it is limited only to the function. Now, if I were to leave the 'var' off and I reload the page, you can see there that now, that variable is accessible. By removing var, I have made that function, that variable global and this is something we do not want to do. So, you want to make sure that you always include var and that you're mindful of what your scope is to prevent unwanted functionality that could give you some problems.

Video 16 – Block Scope

When we use vars, we have no block scope. So, let's consider once again the variable 'age' of '20;'. And then a block with a conditional where we're checking to see if the '(age' of the person is '> 18)'. Then we will create an internal variable where we will hold the string ' 'is adult'; '. Then we will write to the 'console' the '(message);'. Then afterwards we will write to the 'console' '(message);'. Now, if I go ahead and reload the page, you can see there that we write out the 'console.log' in line 6, the very first one, as you can see here. And then we write it out once again on line 9. And that variable, the variable of message, which is declared and assigned on line 5, it's visible from line 9.

That is, we have no block scope when we use vars. However, if we were to use let and const, then we would have block scope. So, I'm going to change 'age' to 'let', I'm going to change 'message' to 'const'. And now, if I try to reload the page, you can see there that we get an error. That message is not defined and the reason for that is that the scope of message is now limited to this block. And in case, block, the word block is not very clear, you can think of it as anything inside of curly braces. So, conditionals, functions, and the like. To illustrate that, let's do one more example where we use a loop.

I'm going to go ahead and create a 'for()' loop here, where my counter is going to be 'i'. I will initialize it at '0;'. Then I will run for as long as the value is less '<5;'. And on each iteration, I will increase by one. Inside of it. All I'm going to do is I'm going to write to the 'console' the value of '(i);'. And then once that loop is complete, I will inspect the value of 'i'. So, inside of here, I will '('check i:' and I will write it to the 'console'. Now, let's go ahead and reload the page and as you can see there, we get 0 through 4, and at the end of the block, you can see there that i is 5. So, let's go ahead and do the same. However, this time we will use block scope.

I'm going to use 'let'. I'm going to start my counter a '10;', and I'm going to run till '<15;'. And then at the end, I'm going to inspect my value and check my value of i. So, I'm going to go ahead and reload the page. And as you can see there, we have the first part, which is the same thing we had before. Now, when we are inside of the block with scope. This one here. You can see there that we run 10 through 14. And then at the end on line 10. You can see there that when we write to the console, we once again get 5. Because the scope of the for loop on line 7 through 9 has disappeared at that point. So, as you can see there, let and const can give you block scope, which can help you control the scope of your variables in a much tighter way.

Video 17 – Closures

Let's create a simple example of a closure. We will create a 'function' that we will call 'outer( )', that will hold an inner function that we will write shortly. Inside of it, we will create a variable that we will call 'counter' and initialize it at '1;'. Then we will write the inner function that we will call 'inner( )' and inside of it, we will write to the 'console' the value of '(counter);' and then we will increase the 'counter++;' every time that function is called. And so, this function that you can see here in lines 5 through 8 can see the counter variable that was declared and assigned on line 3, and at the end of this, we will return the 'inner;' function. Not the execution of the function, but the function itself.

Now, outside of both of these functions, we're going to get a handle on 'inner', by calling 'outer( )', which we'll 'return inner;', as you can see here on line 9, and then we will execute 'inner( );' a few times. So, let's go ahead and do it three times. And let's go ahead and reload that page. As you can see there, the inner function is executing and increasing the counter. And although counter is not part of the function, so you can see here, it has access to it. And not only doesn't have access to it, there is a state that it maintains as well. Now, the very first time you see an example of closure, it might seem a bit odd that you can manipulate a function and a value. In this case, a variable called counter that is not inside of the inner function, not in here, but it is in fact out here.

However, what happens with the closure and the formal way of saying it is that the inner function preserves the scope chain of the enclosing function. And that's a fancy way of saying that the inner function remembers the values of the variables that are in scope. And so, if we were to do this here several more times and let's go ahead and enter it a few more times. We would see that again, that state is maintained, counter is alive. We are increasing it on every call to the inner function. So, as mentioned before, the very first time you see a closure example, it might feel a bit uncomfortable, but this is one that comes up every now and again and it's a good one to practice to make sure that you're understanding is what you think it is.

Video 18 – Closures Vs. Let - Timer Scope

Let's talk about loops. Let's talk about timers, closures, and block scope. Let's start off by creating a loop with a timer. So, I'm going to make a note here, 'loop with timer'. And we're going to create a 'for' loop. I will create a counter called 'i', and I will initialize it at '1'. I will repeat the loop, execute the loop as long as 'i<10;'. And on each loop iteration, I'm going to increase the counter by '1'. Inside of it, I'm going to create a timer. This timer is going to take a callback, a 'function()' that is going to do one thing only. It's going to write to the 'console' the value of '(i)'. And I'm going to schedule this for 1 second. Now, let's go ahead and load this in the browser.

Let's reload the page, and let's see what happens. As you can see there, the value of 10 was written 9 times to the console. And the reason for this is that the loop runs to completion before the timeout executes the timer. Which means that the value of i that is accessible is only the last value that i had, which in this case, is the value of 10. Now, what we want to do is we want to print all the values from 1 to 9. But as you can see here, none of those were written. So, let's try this now with a 'closure'. Let's go ahead and copy this. And now, what we're going to do is that we're going to wrap the timer inside of an anonymous 'function()'. And that function is going to take a parameter that we will call '(x)'.

And inside of here is where we're going to put in the timer. Now, this function is going to be executed immediately. So, we're going to wrap it in a parentheses and then execute it and pass it the value, the current value of i on each iteration. Now, let's go ahead and give the 'console.log' a little bit more information to easily tell them apart. And as you can tell here, we do not need i, but we need x because this is the value that we're taking in, and we're passing in i. And so, I'm going to write here on the very first one, we're going to write first block, in fact, I'll just write ' 'first:' '. In the second one, I'm going to write ' 'second:' '.

And let's take a look and see what happens to the console. So, I'm going to go ahead and reload the page. And as you can see there, we got the first: 10, the first 9 values of 10, just as we did the first time. However, in the second block, notice that we now have all of the values between 1 and9. And the reason for that is that we are creating a closure that brings in the value of i on each of those loops. And so, the value that the timer sees is the value of each of those loops. Now, this works, but it's pretty messy and unless you are very comfortable with closures, we can try for simpler syntax. So, let's try to do this with 'block scope'.

And I'm going to go ahead and paste the same code that I had above, except, in this case, I'm going to use a 'let', everything else will be the same, except here I will write ' 'third:' '. So, let's go ahead and reload the page. And as you can see there, the second is the same as before. And now the third, which is very much the same as the first, with the exception of 'let' instead of 'var' works as well. And the reason for that is that 'let' has 'block scope'. That is, it creates a new memory space on each loop and the value of i that we have is different for each of those timers. And the difference is simply the value of the counter on that iteration. So, this is much simpler syntax. We're leveraging block scope and we do not have to write a closure.

Video 19 – Block Scope Differences - Var, Let, Const

Consider the following block. We have a 'console.log' with a variable called '(i);'. And then we have the declaration and assignment of 'i'. Let's go ahead and load this page on the browser, you can see there that the 'console.log' is undefined. However, that variable does exist. And what happens here is that because 'vars' do not have block scope, i's scope is global. The variable storage space is created, on load. The variable is immediately initialized, and it is set to undefined.

Now, if we used 'let' instead, let me go ahead and copy this block. And we change this here to 'let'. Let me go ahead and call this 'j'. And we reload the page. You can see there that we get an error. We have done the same thing as before. But because we're now using block scope, 'j' does not exist when you reach line eight. That is the variable storage space is only created when you reach the block. And up until that point, the variable remains uninitialized. So, whenever possible it's better to use 'let' and 'const' because it allows you to control scope in a much tighter way.

Video 20 – ES6 Word Count

So, let's give our JavaScript ES6 skills a workout. So, we'll start off with 'lorem', and that's a variable that's equal to a string and this is quite a long string and we'll give you that. First, split that into an array of words so that you can tell me how many words are in this string. Now, then I want you to remove any commas or periods so that now we will have a variable that is an array, but none of the words will have commas or periods. Now, I want you to write a function to display n words per page. So, write this paginate function that takes the data, takes a start location and a pageLength, and returns us a page that we could use, for example, when we're rendering a long list of data.

So, we'll update the start and this will allow us to step through getting pages of data one by one. But for now, just get one page and return it. Then I want you to call it with these arguments. So, we're using the spread operator so that this array will initialize each of these arguments. Then finally, I want you to write a function to count the number of occurrences of each word. You'll find it easiest to do that if you start off with create an empty object. But what we'll do is then fill in so that the name and the value. So, the word will be the key if you like the name. And then we'll have the occurrences of each word.

Check it out by taking your data if your data is called data1 for example. Clone it. So, using the spread function or the spread operator, and then clone it again so that we've duplicated all the data. Now, we should have our words at least twice or more for each word. And now, we should have the count of each word should be two or greater. Okay, so that's a good test on whether your algorithm is working. So, that object should contain basically the results. So, debug that and most of these are one line of code. They are very compact and you will require to use some of the ES6 capabilities.

Video 21 – ES6 Word Count Solution

So, let's warm up with ES6 and do some exercises. So, we're like athletes, we need to get into training. If we haven't been coding for a while, it's good to do this. So, let's take a look. First of all, we're starting out with a string, and we're being asked to do certain manipulations on this string. So, let's load that up. So, we got 'lorem' here. Now, it says how many words are there in the string. So, we need to count those. The easiest way to count them, I think, is to split this long string into an array of words. And we can do that fairly easily. So, let's have 'data = lorem.split'. And we want to split it on spaces. That should do quite nicely. And the ID here is actually quite nice. It tells us there are 69 elements in this array.

Now, we need to get rid of things like full stops, maybe, and commas. Some of these have got commas in like here. Okay, so let's get rid of those. So, one way to get rid of those is for us to do a replace. So, any word that's got a comma in, I want to replace the comma with nothing. So, I need to loop over all the words. Let's call this 'data1 ='. Now, you need to put 'let' 's, if you're coding, and inside the browser. I'm just using the console here to code, but we should do 'let' 's on these or 'vars' or 'const'. But here, I'm just being a little bit lazy. So, 'data1' is going to be equal to, we want to 'map' the data. So, we're going to go over every 'item'. And on each item, I want to do 'replace'.

And let's replace the, oops! Let's replace the ' "," ' with ' " " ', nothings. And that's finished. Replace. Now, this is going to return an array. So, I could just keep going actually, here. Let me get rid of. Now, I want to get rid of, I can do 'map' again on that array, and just repeat this, but this time, get rid of the ' "." '. Now, I'm doing this. I might break this up into several lines, but I want to show you that if you know what's being returned, and I know that 'data.map' returns an array. Then I can operate on that array again and map it again and do a replace. So, let's do that. Okay. So, now 'data1'. Let me scroll up a bit here. And that looks good. So, now we've got a clean array of words and they're asking us now to paginate those words.

So, let's have a look how we could paginate. So, we're going to write a paginate function. So, let me just call it page, 'pageit'. And it's going to be, I'm going to '=>' notation. And we want to start, well, we need the data first. So, we'll, let's take in the data. It's an argument so it won't conflict. It's an argument, so it won't conflict with our global variable called data. And we need to start. So, that'll be where we're starting in the array and we need a 'pageSize'. So, that's what we're feeding in. And now, we'll do a '=>'. And what we want to do is that we're going to, we're going to 'slice' that data. And it tells us we need to put the 'start' and the 'end'. So, the start is the 'start', and the end is 'start+pageSize'.

Okay, that looks good. And I think we're fine for defining that function. Now, we need to apply 'pageit' and input the data. Let me, I just want to show you not a trick, but I'm going to feed in data1 as the start. And let's have the start as say, '3'. And let's have the pageSize as say, '10'. So, it should retrieve one page. And if we wanted to page through all of these, we'd be just calling pageit several times with new start and end positions. But that would be easy to do. So, this is an array. And now, I want to feed that into pageit as an argument or as a series of arguments. So, let me just spread '...args'. See what I'm doing.

Now, I've just got '[data1, 3, 10]' I just wanted to show you, you'll see that you sometimes it's quite nice. And let's see what our result is. Yeah, we get a page one, two, three, four, five, six, seven, eight, nine, 10. And we said we'd start at three. So, now remember zero, one, two, three, sit were zero-based. So, when we say three, we starting at the fourth location. Okay, so that was fairly straightforward. Let's see what we were asked to do now. We've set the arguments, oh, count words. Now, we need to count the number of times each word appears. Let's do that. Let's see if we can do that in. So, I think I'm going to need some, an array, not an array but an object. I'm going to keep track and what I want in my object is we're going to enter the words.

So, we'll have, you know, a word here, and then we'll have the number of times you have to pay us, something like that, right? So, that's what I want to create. And I'll have a whole load of these eventually that we're going to be putting in. However, many would 69, I think, words we've got. So, we'll have 69 of these properties. Okay? So, that's mentally what I'm going to do. List is going to start off as empty. And what I'm going to do is I'm going to do a 'map', and I'm going to get the item. And we need to define, sorry. Now, I'm going to do 'list[item]' because I really want to get the key. I'm going to use. Instead of using the dot, I don't know what the words are like. I can't use that way of getting at properties of an object, but I can use the square brackets, so it's going to evaluate item.

So, here I've got item, it's going to evaluate it. And what I'm going to say is that if it exists, then I want to make 'list[item]'. I want to add one. So, I'll do '++'. Now, if it doesn't exist. I'm going to set it up. So, 'list', it's the first time I've come across the word, then I'm going to set the count to '1'. So, let's see if this will do its work. Well, it seems to, it seems to. Oh, there's one with '2' occurrences there. But most of them are just a singular occurrence. Let's create, let's take our 'data1' and create a 'data2', I'm going to copy 'data1', clone it. How do I clone it? Well, what do we want to do it with an array? So, I want to clone 'data1' and I'm going to put in a copy. So, at least now I'm going to do 'data1' again.

So, at least I've got two, each word is going to appear twice. And maybe it'll be nice to add a couple more words in. Let's have a look. Let's just copy some of these in here. There's a capital letter one here. I think in appears quite a number of times. So, what I'm going to do is I'm going to copy that in here. So, I'm going to add these words as well. So, we're creating a totally new array 'data2' that is going to be considerably longer. So, here we've got it, instead of '(69)'. We've got a '(143)' words now. And a lot of them are scrolled off the page. Now, let's do the word count again. So, in this case, I want to go with data2 instead of data1 and see how many that gives us. Yeah, most of them are '2's.

Ah, I realize that we need to reinitialize 'list' because it's getting added into all the time. So, let's reinitialize it and then run the data2 one again. Here, we go. And now, this is the result of the 'map'. So, it's returning this. So, this is why these are all '1's, it's actually, we're not looking at list. If we look at 'list', that's the thing that tells us. So, here's list, and you'll see it's got '2' of 'Lorem:', 'ipsum:' has got '2', 'dolor:' got '5'. And then it goes on that this is a long. This is a long list. Let's see if we can get the rest of them. Yeah, so this is the whole list. So, '3's and most of '2's because we duplicated the array. So, this is the whole array of how many word counts. And you'll notice this 'ut:' has got '4', most of '2's because we just duplicated. Some have '3'. That's our word count, and it's pretty neat.