December 7, 2021

JavaScript Syntax and Operators

**Modules in This Course**

-Switch statement:

-simplifying multiple if-else statements,

- block-level scope issue: how to resolve that

-Difference between for/in and for/of loops.

- using the appropriate for loop for the appropriate condition

- break, continue, and labels

-Using Math and Comparison Operators:

- demos of the different operators

- use strict

- Working with Logical Operators and Short-circuit evaluation.

- truthy and falsy

- short-circuit evaluation works.

-Utilizing JavaScript Exception Handling

- try…catch

- blocks

-How to Determine JavaScript Variable Data Types.

- use the typeof operator

- instanceof operator to see how to determine what type of data you're working with within a variable.

-Understanding 'this' in JavaScript

- the use of 'this' in different scopes

- Call () and apply () methods

-Using powerful spread operator

- manipulate arrays

- pass arrays to functions.

**How the Switch Statement Works**

**The switch statement**

**Text

Description automatically generated**

-Use the switch instead of multiple if/else statements.

-simplify our code a little bit.

-use ‘**case**’ statements to compare to each expression in the switch statement

-‘**break**’ statements to exit out of each case

-the default statement is used when none of the cases match.

I.E., **a switch statement here that has some expression: compare the value of that expression against each one of the case statements that we have, and you can have one or many case statements. That's the reason why you would use a switch**. Once a statement matches up, we'll execute a statement or a series of statements. You can have as many statements as you wish within each case statement.

Timeline

Description automatically generated with medium confidence

very important is, you have to have a **break statement** before the next case because the break is what will then say, continue with the line after the complete switch block. Now if no other cases match, then we use **default**. So if none of the other cases match up to the expression's value, then any of the statements within the default will execute.

**Simple Switch Statement Demo**

I.E., simple switch statement, and default statement that I talked about can actually be placed anywhere. It doesn't have to be at the bottom of the switch. I'm using Visual Studio Code, and I created an HTML document with a button, as you see on line 9, that in its onclick will call a function called Simple Switch that is already created on lines 11 through 13. Now what I'm going to do is, I'm going to go ahead and declare a variable in here, like so, and I'm going to use this variable as the expression for my switch statement. So in the switch, I put in some expression. The expression just happens to be whatever this variable's value is. Now within the switch, we put our different case statements. So you can see I'm going to do a case 1, and I'm just going to simply spit out Product 1 to the console. And then don't forget your break. We have to have the break because that's what separates each individual case statement. So we can do Product 1, Product 2, break, and then if none of those match, we can do a default. And on default, I'll just simply say, Unknown product, like so. So I save this HTML document, and what it's going to do is, it's going to run this code, it's going to set productId to a 2. It will look at the value of productId and take the appropriate case statement. So let's go ahead and open up in our default browser. Let's bring up our F12 tools and click on the Switch Sample button. You can see down in the console, we get Product 2 showing up. Now, if I go back and I change this to a 1, save this, go back to the browser, refresh, we now see Product 1. And if I go back and change this to a 3, save, go back to the browser, refresh, we should now see Unknown product. So there's a very simple little switch statement. One thing I did mention is that the default can be anywhere. It doesn't have to be at the end. You could put it at the beginning or anywhere. Kind of the more style that most people do is to put it at the end, but you can put it here. If we then go back to our browser, refresh, you can see we get the exact same result.

Multiple Case Statements and Forget a Break

One of text things you're allowed to do with switch statements is, you're allowed to have multiple case statements. So what happens is, if the expression that we're testing in our switch matches any of the cases, it'll fall through and execute the statement or the statements. And then we hit the break, so then it would break out of the switch. So this is perfectly legal, and we do this when we do want to match up an expression on many different types of cases. So, let's take a look at multiple case statements, and let's also take a look at what happens when you forget a break. As you can see, I've added a new function here called multipleCase, and I have a button then that will call this function. On line 13 we set a variable named color equal to the value Red. On line 15 is where we're going to switch on that variable color. Now notice lines 16 and 17 where I do a case of Red or a case of Pink, I then, I'll console.log, The color is red, and then I do the break. So what it does is, it checks that color against both red and pink, and if either one of them match, it then drops into the line 18. You can then see a couple more cases where I've got a Blue, a Light Blue, and a Dark Blue. And then we do the console.log, The color is blue, and then a break. Lines 25 and 26, we have case Gray and Grey and console.log, The color is grey, and then again, a break. Let's go ahead and run this really quick so we can take a look. Bring up our F12 tools, and you can see that indeed we fall through to the correct place, which is, The color is red. Let's now take a look at our second example, which is on this button click, where we forget a break. We call this method here, so you can see on line 37 we create a productId variable, set it equal to 2. The switch on line 39 is checking that expression. Now, look at the case 1. That one's fine. It has a console.log and then a break, but look at case 2 where we just have a console.log, and we have forgotten the break statement. Then we have a case 3. Notice what it is, console.log Mountain Bike Socks, and then we break out of that. Now, let's go ahead and run this and click on this. So what happens when you forget a break statement is, it simply falls through to the next case. So even though it matched up on the case of productId of 2, it still is going to, since there's no break, it's going to continue on with the next executable statements, which, in this case, happens to the be statements within the case 3. Because that's where then the next break is, it then falls out of the switch.

**Switch Uses Strict Comparison**

Now the switch statement does what's called a strict comparison. What that means is that the type and the value must match in order for the expression to match to one of the case statements. Let's take a look. We have a new function called strictComparison on line 11, and the button on line 9 actually calls that. You can see on line 12 I set a new variable, productId, but look what I did. I set it to the value 2, but that's a string. Then within the switch statement, notice lines 15, 18, and 21 are a case of 1, numeric, 2, numeric, and 3, numeric. What do you think's going to happen if we were to run this code? We get Unknown product because both the type and the value did not match any one of those case statements because the value would've matched line 18, but the type doesn't match. As a result, none of the cases match the expression. It then falls into the default statement, which prints out Unknown Product.

Block-level Scope Issue with Switch Statements

As you probably know, in JavaScript, as in C# and as in a lot of other languages, there can be a block. And a block is enclosed by the curly braces in these types of languages. So the switch statement is a block. But each case statement is not a block. If you want to make a block, you must make statements a block by wrapping them in braces. So the case statement can be wrapped within braces, and that would then make it a block. So why is that important? Well, let's take a look at a demo, and I'll show you the block-level scope problem and a resolution to that problem. On line 12 I have a function called blockScopeProblem. On line 13 I declare a variable, productId, and assign it equal to 2. Now, take a look down in case number 1 where inside of there, I try to declare a variable called message, so let message equal to, and then this string value, and then a console.log on message. So then in case 2, I also declare a variable, let message equal to this string, console.log message. And then I do the same thing on line 25 within case 3 and on line 29 within the default. So as I mentioned before, the case statements themselves are not a block. So what would happen if we were trying to declare message in each one of these case statements? Let's open this up in our default browser. Let's hit F12, and look at this, we get immediately a message that says Identifier message has already been declared. That's because each one of these is not a block, so it thinks that we're just declaring this variable over and over again, and of course that's not allowed. So how do we fix this problem? It's actually very easy. Here you can see on each case statement, so case 1, I have added the brace and then a closing brace, and then I declare message inside of that brace. By making the braces, that creates a block. I then do the same thing for number 2, case 2, where I add an open brace and a closing brace and then I declare the message within here. This is now legal syntax. Now each of these message variables are declared within a block, and then they are thrown away once the block ends. This is how you can fix the block-level scope problem within a switch statement.

**Module Summary**

In this module, we saw how the switch statement really helps us with our readability. It's also more efficient than using multiple if/else statements. The only thing we need to really kind of be careful of is with this block-level scope, but I showed you an easy fix for that. Coming up in the next module, we're going to talk about using the appropriate for loop. We'll also talk about break, continue, and labels. I hope you'll join me for the next module.

**The Difference Between for/in and for/of**

**Using a for/in Loop**

This module is the difference between for/in and for/of. The goals for this particular module are to look at a for/in statement and a for/of statement and talk about the difference between those two. We'll also take a look at the break statement, continue, and labeled statements. Let's get started. Let's start out looking at the for/in statement, which iterates over the elements of an object. These are the properties and the methods. Now each time through the loop, it's going to return the key, or the property or method name. You can then use that object and then use the square brackets with the key name that you get back inside of it, and that will return the value to you. Let's take a look at using the for/in loop. On line 9, you can see a button that calls a forinSample. This sample function is down here on line 15. Now you see on line 16 through 26 I declare a literal object called product, and inside of that, I have several properties such as productID, name, productNumber. I then have a method called calculateGrossProfit. Now, if we take a look down here on lines 28 through 30, you see a for loop. The key here is that we use this "in" keyword. So we're saying for const key in product. So the "in" says iterate over the collection of properties and methods inside of this literal object called product and return the key to me. Now you can see on line 29 I can spit out the key, and I can access the product and use that subscript, the square brackets key, which is a subscript into that to retrieve the value. Let's go ahead and run this, bring up our F12 tools here, and click on the button. So you can see, productID is the key, and then it's equal to 680. So we're reporting the actual property name and the value. Then look at the very bottom here where we have calculateGrossProfit, so it actually returns the method to us and shows us the body of the method as well. So this is a for/in loop.

**Using a for/of loop**

Let's take a look at a for/of loop. What this one does is iterates over the values in any iterable object, such as an array or a string. It returns then an object for each iteration. What the object is depends on the type of object you're iterating over. Let's take a look at a couple of examples here. We'll do a normal for/of where we iterate over an array, and then we will actually loop over a string. On line 9 I have a button that calls a function called forofSample. If we scroll down and we take a look, here's this simple little function on lines 42 to 46. What we're going to do is iterate over an array of products, and that's declared right here, let \_products equal to an array, and it's an array of objects. So I've got a couple of different product objects here. On line 43, we say for const item of, so we're using the "of" keyword now. So the "of" says, hey, let's now iterate over this iterable object, which, in this case, is an array. So each time through the loop, it's going to place each item of the array into that constant called item. And then on line 44, I'm simply going to do a console.log, I'll do a JSON.stringify so we can output that as a string, that object as a string, to the console. Let's go ahead and view this in our default browser, bring up the F12 tools, and there we can see the iteration over the array of three items, each time through the loop spitting out the stringified value of that object into the console. Our next example now loops over a string. So on line 9 I've got a button which, when clicked, calls loopStringSample. That function is on line 15. Inside of this function on line 16 I let productName equal to, and then you can see this string of characters here. And then I also have another let on line 17, letters = empty string. What we're going to do is, we're going to loop through now for const character of productName. So it's treating the string now as an iterable object, so each time through it's going to be a single character of the string. We're then going to take that character and add it to the letters variable that I've created, and at the very end on line 22, we'll spit out letters. Let's go ahead and view this in the default browser here, bring up our console, and basically what we get back is the same string. The difference is we iterated over that string one character at a time. You probably won't do this too much, but it's nice to know that a string is an iterable object.

**Break and Continue Statements**

Let's take a look at two other statements that you might use in combination with looping. First one is break. Now, we've seen break before. That was in the context of the switch statement. In this case, what break does is leaves a loop early. The other statement you might use is continue, and continue will let you go back up to the top of the loop and continue with the next iteration, bypassing any code below the continue. Let's take a look at these two demos of break and continue. Our first example is a break. I still have the same products array. But down here now you can see the function breakSample. I'm going to now use the for loop, and I'm going to do a for/of because I'm iterating over an array of product objects. What I'm going to do is, I'm going to check inside of the for/of. I'm going to do an if statement to check to see if the standardCost is less than 20. If it is, I will break out of the loop, which means it skips any other statements below the break. On line 47, what we're going to do each time through the loop is do a console.log on the JSON.stringify of the item. But once the standardCost is less than 20, we will break out of the loop. Let's take a look up here, and we see that the first standardCost on line 20 is 1059. But then on line 28, the standardCost is 13.08. So it should skip the last two items in the array. Let's open this up in our default browser, bring up the F12 tools, and we see that indeed we do just get the first object and then the rest of the loop is broken out of. The continue statement in our continueSample, if we take a look at this function, this one is kind of the opposite of break. What this one will do is go back up to the top of the loop and continue with the iteration but still skip any lines below it. So just like the break skipped any lines, this one also skips them, but the difference is, it goes back to the top of the loop and continues. So on this one, inside of our for loop, we have an if statement that checks to see if the standardCost of that product object is greater than 1000, and if it does, it says continue. So the first one is definitely over 1000, so it will not perform line 47. Instead it will go to the top, get the next item, and then continue. Let's go ahead and open this up in our default browser. We'll bring up our F12 tools, click on continue, and we should see only two products get reported. The first one is skipped.

**Labeled Statements**

The last statement I'm going to cover here is called a labeled statement, and that this does is define a location to go to. Now, anybody who remembers the old basic days of goto, you know this was a very bad idea. It's still a bad idea in JavaScript. It's not recommended for use. But I wanted to show it to you just for completeness and because, you never know, you may inherit some code where somebody actually did this. So you know not to use it, but somebody else may not know. Let's take a look at labeled statement. On line 9 we have a button that calls a function called labelSample. So here's the function labelSample on line 16, and then notice on line 17, there's something with a colon after it. Now we've only seen that in case statements before, but this is actually a label. A label is whatever name you want, I call it even, with a colon after it. What I'm going to do is loop through from 1 to 10, but I'm going to check that index number every single time and see if it divides evenly by 2. If it does, then we do continue even. When you put a label after the continue statement, it doesn't go to the top of the loop; it actually goes to that label. It then drops back into the loop and continues on. So what this'll do is report back only the even numbers. So let's go ahead and click on this, and we should see just the even numbers appear. So as you can see, a goto is not a very good construct to use. It leads to spaghetti code. You kind of never know where you're going to go back to. There's always a way to refactor your code so that you never have to use a labeled statement.

**Module Summary**

In this module, we took a look at some specialized for loops for iteration. We used a for/in for iterating over objects' properties and methods, and we used for/of for any other iterable objects such as arrays or strings. We saw how the break and continue statements help us control the flow of a for loop, and we learned that a label is a goto mechanism, and again, we should avoid this at all costs. Coming up in the next module, we'll talk about math and comparison operators, and we'll find out what that use strict statement really does in your JavaScript functions. I hope you'll join me for the next module.

**Using Math and Comparison Operators**

**Mathematical Operators**

This module is Using Math and Comparison Operators. The goal for this module is to look at the various JavaScript operators such as math, assignment, and comparison. We'll also take a look at using the plus sign both with string data types and numeric data types. And we'll talk about use strict and what this actually does in your JavaScript functions. Let's get started. Just like most programming languages, JavaScript also has a large set of mathematical operators. We have addition, represented by the plus sign, subtraction; which is a minus sign, or a dash; we have multiplication, which is the asterisk; we have a forward slash, which is division; exponentiation is two asterisks, we have modulus; which is the percent sign, we also have an increment and a decrement. So let's take a look at math operators in a JavaScript function. On line 9 we have a button that calls a function called mathSample. We see that function here on line 14. I've got a couple of variables, price and result. I've set price to be 200. And we're just going to simply perform some math operations and then put it out to the console window. Let's take a look at addition. Addition of course is a plus sign. So we're going to take price + 100, put it into the variable result, and then we'll spit that out to the console.log. Subtraction, we'll do result = price - 10, and then we'll put that out to the console. Multiplication, we'll do price \* 2, so again, we'll place that into the console. Let's then take a look at division, which is the forward slash, so price / 2. We have exponentiation, where we're multiplying price to the power of 2. And then of course modulus, so price % 3. Let's take a look at the increment and decrement. So these two are a little different just because where you place the ++ or the -- makes a difference. So line 45, result = price++. When you put the ++ after the variable, it takes the value of the variable and assigns to result, and then increments the price, so we will see that in the console.log. And then if you do result = ++price on line 48, what we get there is, it increments the variable first and then assigns it to result. The same thing for decrement. If you look down on line 52, result = price--, it's going to take the value of price and assign it to result first then decrement the variable price. Let's go ahead and run this and take a look. So you can see all the various addition, subtraction, multiplication, division, exponentiation, and then modulus, and then you can take a look and see what happened with the increment and decrement. So notice result = price++ = 200. When we're spitting result, remember, price was set to 200. So result is 200, but then immediately after, if we print out the value of price, we see that it was incremented after that line to 201. If we then do result = ++price, it now is equal to 202 because it increments price first and then assigns it to result. We can then do the same thing with the --, and we'll see the same exact functionality happening there as well.

**Plus Sign with Strings and Numbers**

Let's talk about the plus sign being used with string and numeric data types. Now the plus sign is overloaded, so when you're using the string data type, the plus sign is used to concatenate two strings together. When you're using numeric data types, it is addition. So what happens if one is a number and one is a string? So for example, if we do result = 100 + 200? What happens? Well, if one is a string, it's going to do concatenation. So you have to really look at both operands here before you figure out what's really going to happen. So let's take a look at the plus sign. So in this sample, we have a function called plusSignSample that's called from the button on line 9. On line 14, create a variable called price, set it equal to a numeric value 200. We're going to create a variable string value and assign it equal to the string 100. And of course we'll have a result variable as well. Now, what we're going to do is, on line 20 we'll say result = price + stringValue. What JavaScript does is, it looks at both sides and figures out, are both numeric? In this case, they're not. The string value is data type of string. So it's going to do string concatenation. It will convert price to a string and then concatenate the two together. Now, there's one more trick that we can do here. If you know you have a string value and you wish to convert it to a numeric, line 23, you can see what I've done, result = price +, and then put a plus sign in front of the variable that is the string value. That will convert it to a numeric. Let's go ahead and run this. And we can see the results. The first one performs the string concatenation. The second one, since we converted the string value to a number, performs addition.

**Assignment Operators**

Just like in C#, there are assignment operators. Now, we know the equals sign assigns one value to another, but we also have this shorthand for doing addition, where if you wanted to do price = price +5, you can do this shorthand of +=. And we can do that for subtraction, multiplication, division, exponentiation, and modulus. So these make it very simple, just a nice, little shorthand instead of having to type so much. This gives us kind of a more efficient way to express these. So let's take a look at a demonstration of these assignment operators. We have a function called assignmentSample. I've again created a price variable and set it equal to the value 200. We can now do the addition, where we do price += 100, and then we'll spit that value out. We can do a subtraction of price -= 100. We can do a multiplication of price \*= 2 and a division of price /= 2. We can also do the exponentiation and the modulus. So these are just nice, little shorthands. Let's go ahead and run this, and we'll take a look at the result. And it is nothing spectacular. It's something we all should just know from basic math.

**Comparison and Ternary Operators**

A lot of times in if statements we need to do some comparisons, so we have all the standard ones you'd come to expect in any language. We have less than. We have less than or equal to. We have greater than. We have greater than or equal to. We have equal in value, and we also have equal in value and type. And this is the one you should use most often when you want to actually do real comparisons. So it's triple equals. We have not equal in value, and we have not equal in value and in type. So let's go ahead and take a look at these various comparison operators. We have a function called comparisonSample where I've set the price equal to 200, and then we're going to take a look at are all the various ways that we can do comparisons. So, on line 21, we're going to actually check to see if price is double equal price. Now, of course, that should come out to be true. But then look at the line 24. I'm going to set result now equal to price.toString, so I'm changing price to a string. When we use the double equals to compare the price, which is a number, to the result, which is a string, this one should come out true because we're using double equals, which just simply checks for value. So what it does is it looks inside and figures out if that string can actually be converted to a numeric and then performs the comparison. Now let's take a look at the next one where I set result equal to price. So what happens is now when I'm using triple equals, it's saying is price and result the same data type and the same value. That should come out to be true. But then look at the next one on line 32 where I again set result equal to price.toString. So now we're comparing a price, which is a numeric, to a result, which is a string, and we're using triple equals, which means the value part, yes, that's true, but the data type is not, so that one is going to report a false. We then have the same thing for not equal. So on the not equal, we can do a exclamation point equals, and it'll check the value, or we can do an exclamation point double equals, so that gives us the three, just like the three equal signs, but we're using an exclamation point for the not, so if they're not equal in type and value. And, of course, we have all the rest of them, greater, greater than or equal to, less than, less than or equal to. And then, of course, we have the ternary operator as well. What this says is if the first part of this expression, which in this case is price less than 1500, question mark means take the next piece between the question mark and the colon and report that back. The colon says this is like the else portion. So if price is greater than or equal to 1500, then this part would be spit out, which would be greater than 1500. Let's go ahead and run this, and we can then see the results, which are the same as what we discussed when we were looking at the code.

**The Effect of 'use strict'**

Use strict is a statement that you should always be using. So let's take a look at what it really does. Now first off, you've got to understand that use strict is ignored by any older browsers. Because it's in quotes here, it's just simply an expression. What we use this for is to force all of our variables to be declared before they are used. So mistyped variable names actually are created, globally scoped, if you are not using strict. And there are a few other rules as well, which we'll talk about as we go through the demo of use strict. In this function useStrictSample, on line 14 I have the use strict. Now remember, you enclose this in single quotes, and that way, it's just an expression, and older browsers can then ignore it. Now, when you have this set, you can't do things like create a variable without using var or the let or the constant or the keywords. So line 17 won't work. It'll actually bomb on us. Let's actually take a look. So you can see we get an uncaught reference here, result is not defined. That's because the use strict is on. If we go back and we comment this out, save this, refresh, and run it again, it now works. So the use strict forces us to declare variables, which is always a good practice, as it is in any programming language. Now as I mentioned, there are a few others rules as well. For example, you can't use reserved words as variables, so eval or arguments. You can't delete a variable, and you can't delete a function when you have use strict in effect. However, if you don't have use strict in effect, you can do all of these things. Again, best practice, always use strict.

**Module Summary**

In this module, we learned to recognize the different operators such as math, assignment, and comparison. We saw the effects of numbers and the string data type when using the plus sign. And very important, we learned that we should always use the use strict expression. This will help keep our applications running at tip-top form. Coming up in the next module, we'll talk about logical operators, truthy and falsy, and we'll see how short-circuit evaluation works. I hope you'll join me for the next module.

**Working with Logical Operators and Short-circuit Evaluation**

**Truthy and Falsy**

This module is Working with Logical Operators and Short-circuit Evaluation. The goal : learn about true and false values. Now those are known as truthy and falsy in the JavaScript world. We'll talk about learning the logical operators that are also available, such as and, or, and not. And we'll take a look at this concept called short circuiting. Let's get started. In JavaScript, any variable with a value, such as a string that has characters in it, a numeric that is non-zero, or a Boolean true is considered true, and you can use it in an if statement like that. Any variable that is a Boolean false, null, undefined, not a number, or an empty string is considered false. So, let's take a look at an example of how this truthy and falsy works. Let's take a look at this function called trueFalseSample. I have two variables, price and color. I've set price to 200 and color to Red. Again, based on what we saw in the slides, since price is something other than 0, it's considered a true value. And color, since it has some values in the string, is also considered true. We can check that by, on line 22, we can say result = price greater than 10, so obviously that evaluates to true. So result when it prints out will be true. We can also use, on line 26, if price is greater than 10, we can test the expression in an if statement. But we can also simply do this, on line 31. If we just care whether or not price has something in it, we can just simply say, if price. We don't need to say greater than 0, because again, if it has a value in it, it's considered truthy. Same thing on line 36 here, if color. So again, we don't have to check to see if anything's in it, or we don't have to do if color.length, we can simply check if color. And if so, it will print out color has a value. So let's take a look first off at these. Let's go ahead and open this up in our default browser and take a look at the truthy values. So up here at the top, we can see, the first one is if price is greater than 10, that's true. And then we get the other console.logs that say price is greater than 10, price is greater than 0, and color has a value. Let's go back and take a look now at the false values. So now, as you can see on line 43, color = null. So if I were then to take color and put that into the Boolean function, which actually says take this value and convert it to a Boolean. Now, you don't have to do this. I'm doing it because I'm not using an if statement; I'm just doing it so that we can print out the value in the console. But this will come out false. Color = empty string is false. Line 51, color = undefined will also become false. If on line 55 you let value and you don't assign anything to value, that will also be a false value. If you end up doing some arithmetic and that comes out to be not a number, that will also become false. Again, we could just simply run this, and now we can see all of these values down below are now reporting false values.

**Logical Operators**

Let's take a look at some logical operators now. So we have the logical operator and, which is a double ampersand. So an example would be price is greater than 10 and price is less than 1600. We have an or, which is the two vertical pipes, so price greater than 10 or price is less than 1400. We also have the not operator, which is an exclamation point. So this will negate anything, so not, and then price greater than 10. So if price was equal to 200, then this would come out to be a false. Let's take a look at our various logical operators in action. Let's take a look at this function logicalSample. On line 16 I let price = 200. On line 19 I say if price is greater than 10 and price is less than 1600. So as long as both sides of that and are true, then the whole expression is true. So on the console.log, we'll print out that string. Now take a look at line 24. If price is greater than 10 and price is less than 200. Well, obviously both sides are not true since price is equal to 200, thus, the whole statement is false. On line 32, if price is greater than 10 or price is less than 1600. So in this one, if at least one of the expressions is true on each side of the or operator, then the whole expression is true. Take for instance line 37. If price is greater than 10, it is, or price is greater than 1600. It is not. However, because one of them succeeds, the whole expression is true. Then on line 42, if not price less than 10. So it evaluates, and since the price less than 10 is within parentheses, that piece is evaluated, and then the not operation is applied. So is price less than 10? It isn't; it's 200, so the expressions is false. We apply the not; it becomes true. Let's go ahead and open this up in our default browser. Let's click on the Logical Operators button, and we see the results just as I explained in the code.

**Short Circuiting**

Let's now talk about short circuiting. Now short circuiting is simply an optimization for logical expressions. What it allows JavaScript to do is to bypass subsequent expressions in And or Or conditions based on truthy or falsy. So let's take a look. So here's an example where I have an isColorRed function, and this function, if I pass in a Black, it's going to return a false. Then as you can see, I'm calling in the same expression another function called isGreaterThan1400, and I'm passing in 1401. So that will actually return a true. We're doing an and. Now as you remember, an and says both sides must equate to true. We know that the first one is not going to. So when we're using the and operator, if both sides must return true but the first one doesn't, well, why should it even run that second expression? And that's exactly what happens. And that is short circuiting. Now, short circuiting for the or is different than the and because when using the or operator, only one expression needs to be true for the whole expression to be true. So if we take this example now where we say isColorRed and pass in Red, that's true. Thus, it does not need to run the other side of the or because right then the expression is now true, so the isGreaterThan1400 does not get run. So, let's take a look at a demo here of short circuiting. Here we can see the function shortCircuitAndSample, so we're going to use the and. Now look down on lines 23 through 26, and you see the function isColorRed. And you can see what we're doing. We're simply return value === Red. And I'm going to put a console.log in there because I want to show whenever it goes into this function. Then on lines 28 through 31, I've got the function isGreaterThan 1400, again, passing in the value, and we're checking to see if that value is greater than 1400. Again, I'm using console.log to show that we've gone into the function. Now, go back to line 20. Result = isColorRed but passing in Black, and, so we're using the and operator here, isGreaterThan1400. So what does this mean? It means that since we're doing the and, JavaScript is going to first evaluate the isColorRed("Black"). It will come back false, so it never calls isGreaterThan1400. If we've done this correctly and if JavaScript is evaluating this correctly, all we should see in the console is this message. Let's go ahead and open this in the default browser, bring up our F12 tools, and we can see that yes, indeed, short circuiting is taking effect here. Let's now go take a look at the short circuiting using or. As you remember in or, each expression is evaluated until one returns true. As soon as one returns true, none of the rest of the expressions are evaluated. Take a look at line 41. We call the isColorRed, passing in Red. Now, as we know, that will actually return a true. Thus, there's no reason for it to continue on and do the isGreaterThan1400. Take a look at line 47. We then switch these two around. We do the isGreaterThan1400, passing in 200, which, as you know, will now return a false. So it now needs to go to the other side of the or and now do the isColorRed to see if that one will then return a true. Let's go ahead and run this. And now we can see on the first example it calls the isColorRed first. That one returns a true. It never goes in to the isGreaterThan1400. On the second one, however, as you can see, it calls the GreaterThan1400. Since that returns a false, it must go to the other side of the or and test that expression, so it goes into the isColorRed. And this is how the short circuiting for or works.

**Operator Precedence**

We have one more thing to talk about when we're talking about all the different operators, and that's order of precedence, so which order things get evaluated in when you have a big, long, complicated expression. Well, obviously grouping first, so you can group things by using parentheses, then not, then multiplication, then division, then modulus, then addition, then subtraction. So all of these arithmetic ones happen in the order they occur, left to right. We then have less than, less than or equal to, greater than, greater than or equal to, equal, not equal, strict equal, strict not equal. We then have and, or, and then finally assignment. So there is an order to which things occur. So you can look this up on the internet. There's a lot of people that have put some really good examples out there. It would be a good way to test your knowledge. In this module, we learned about truthy and falsy. True is a Boolean true or any variable that has some sort of value, whereas false is false, null, undefined, not a number, and an empty string. Logical operators and, or, and not help you make complicated decisions, but remember short circuiting because that's going to determine if one side of that expressions runs. And if that side of the expression is a function, that may or may not run that function. Coming up in the next module, we're going to talk about exception handling. We'll talk about try…catch and finally. I hope you'll join me for the next module.

**Utilizing JavaScript Exception Handling**

**Try, Catch, and Finally Blocks**

Hello everybody, Paul Sheriff here with Pluralsight. This module is Utilizing JavaScript Exception Handling. The goals for this module are to learn how to handle exceptions using the try…catch statements and the finally statement. We'll also talk about how to throw your own custom exception and how to determine the type of error that has just been thrown. Let's get started. Your basic design pattern for when running code that could fail is to wrap it inside of a try…catch block. So we put the code that could possibly fail inside of the try. Then if something does fail, it jumps down to the code within the catch block. So an error object is created, and it's passed to the catch block. Now the finally is code that runs every single time. This is optional. You may not need a finally block every single time. Let's take a look at a demo of a simple try…catch block and also add on a finally block. In this function simpleTryCatch, we declare a variable called result. We're then going to do in a try block result = x / 10. Now this code will actually error. JavaScript creates an error object with a name and a message property and passes that to the catch block. Now you can do whatever you want to do within the catch block. I'm going to simply output the message to the console. Let's go ahead and run this, and we'll take a look at how this works. So there you can see the error message that JavaScript created for us, x is not defined. Now what you decide to do with that error is completely up to you. Some people might publish this back to a server. You may even log it locally if you want to, although you really need to get it back to a server at some point. That's completely up to you. But let's take a look at the finally. So here's another function called finallyCatchSample. We're going to do basically the same code. We're going to let result. In the try block we're going to do result = x / 10. Now I'm adding some additional console.log messages in here just to show you what will happen. So when an error occurs on line 40, line 41, or any other lines after the line where the error occurred, will never run. We then dive into the catch block. That code runs, and then if you have a finally, that code will always run, even it goes through the try block normally or if it goes through the catch block. Let's go ahead and run this. And as you can see, an error will occur. We then go into the catch block, I also spit out that error message again, and then we're in the finally block. So when an error occurs, we go into the finally. However, if it succeeds, so now let's go ahead and declare x, so x is now defined, and now we'll see that we will also go into the finally block even on success. So we'll go ahead and run this. And now you can see an error won't occur, that was the console.log statement, but no catch block because everything went successfully. But the finally block still runs as normal.

**Throw a Custom Error Object**

While JavaScript gives us an error object, a lot of times the messages are just a little bit too cryptic and maybe don't explain enough about where you are in your code to help you track down the error. If that's the case, you can throw your own custom error object. And you might want to do this quite a bit. What you do is create an object with at least two properties, message and name. Then you can fill in the name with whatever you want and the message with whatever you want. And the message could even be like what file you're in, what function you're in, what object you're in, and include other properties of the object even that you happen to be in. There are all sorts of things you can do with this custom error object. So let's take a look at throwing your own custom error object. It's a simple one, but will give you the idea of how to at least start this process. I have a function called throwError. In this, I try to call the attemptDivision, and I do that in a try…catch block. If we take a look at attemptDivision, it's basically the same function we saw before. We let result, we try result = x / 10. But inside of the catch block now, once we catch that error, instead of just grabbing the original error message, I'm going to throw my own custom message because this way, I'll be able to add on some additional information. So look at the throw, this object where I've got a message and a name property. Now, in the message, we say In the attemptDivision method the following error occurred. And I still grab the original message. That's always good to have. But I've added on some additional information here to make it a little clearer about where I was when the error occurred. As far as the name, you can give it any name you want, but I would try to give it something that maybe is a little unique. I just use CustomError here, but maybe you want to do CustomAttemptDivisionError or something like that, something unique and something different than the JavaScript errors, which I'll show you in just a minute. But let's go ahead and run this. So let's throw our custom error, and there you see the error gets bubbled back up. We then display that error message, and I added on also the Error Type. So let's go ahead and go back here, and we can see right here, after I've displayed the error message that came back, I then display the Error Type, which is the error.name property.

**Detect the Error Type**

Now we saw reporting the error type by using the error.name. Well, let's take a look at the built-in errors that get thrown by JavaScript. There is a ReferenceError, a RangeError, a TypeError, a URIError, a SyntaxError, and EvalError. Now EvalError is for backwards compatibility only. It's not used anymore. But you may still see it in some older code. Let's take a look at determining the error type. In my sample here, I've created a function called handleError, to which I'm going to pass the error object generated by JavaScript. And then on line 73 I'm going to switch on that error.name property. Because JavaScript is consistent, it will always pass me the same error name for each of its built-in error objects. So you can see we have a case for ReferenceError, for RangeError, for TypeError, for URIError, for SyntaxError, and for EvalError. Now, if you have your own custom types, that would then fall into the default, or you could add additional case statements here as needed. Let's then go all the way back up here to the top, and we can take a look. So our reference error is what I showed you before. The reference error is when you have a variable that is not defined. A range error is because you did something that caused an error in a numeric value. So in this case, I'm trying to do a toPrecision, which is a built-in JavaScript function that tries to give you 200 significant digits. Well, that's not possible according to JavaScript, so a RangeError occurs. Type error happens because the result is a numeric, and we're trying to apply a toUpperCase to a numeric value. Again, that doesn't work; that's only for strings. URI error happens when somebody gives you maybe a URL and it has some invalid characters in it. And you try to maybe decode that URI; it will throw an error to you. Syntax error, this is the one that's usually used now instead of EvalError, so if you're using the eval function, and, in this case, I forgot the missing single quote on the end of Hello there, so when you're trying to do an eval and you get an error, you're going to get a SyntaxError. And then as you can see, in each one of these catch blocks, I simply call the handleError. Now that's just this sample. You can do whatever you want again, but it's nice to know that you can check for the type of error and then decide what to do based on that error. So we can go ahead and run this just so you can see it. We can see a reference type and what its default message is, we can see a range error and what its message is, we can see result.toUpperCase is not a function, we can see that the URI is malformed, and we can see when we're doing the eval that an invalid or unexpected token occurred. And you'll get all sorts of different error messages with that syntax error. But these are an example of the built-in JavaScript error objects.

**Module Summary**

In this module, we saw that we should always add a try…catch around dangerous code, so anything that could cause an error, let's wrap it in that try…catch. You don't always need a finally block, but if you do need it, it's there, and remember that it runs every single time based on we go through normally or if we go through the catch block. So that finally block will run. We can also throw custom errors to communicate any specific information. This is very handy to give you additional information that you may not get just from the normal JavaScript error object. Remember to always include a name and a message property. Also, you could change how you handle errors based on the type of error. So you saw how I was passing in that error object to a switch statement, and then inside of each case, I could do a lot of different things based on the type of error. Coming up in the next module, how to determine data types. We'll talk about the typeof operator and the instanceof operator. I hope you'll join me for the next module.

**How to Determine JavaScript Variable Data Types**

**Data Types and Detection Methods**

This module is How to Determine JavaScript Variable Data Types. The goals for this module are to explore the different data types that are available, and these include primitives and objects. We're going to learn how to determine data types using the typeof operator, a constructor property, and the instanceof operator. Let's get started. JavaScript, unlike other languages, only has a few primitive data types built in. It has Boolean, which can be true or false; null; which is no value; undefined, which means you've declared a variable but it has no value; and a number, integers, decimals, float, etc., all are represented by number; and string, string is a series, an array, of characters. These are the only primitive data types that are available in JavaScript. There are a few object data types as well, such as an array, so you build it with new Array, and that's a collection of values. There's an error, which contains a name and an error message. There's a function, which is a block of code. There's an object, which is a wrapper around any of the other data types, and RegExp, so a regular expression. These are the built-in object data types. We also have these that are similar to the primitives, which is a new Boolean, so very similar to the primitive Boolean, but it is an object that contains true or false, new Number, so a number is an object that contains a numeric value, again, similar to number, or new String, an object that contains a character or a series of characters. Now all of these are like the primitives. So, what I want you to do make sure you use the primitives, Boolean, Number, or String, instead of these whenever possible. These types of objects take up more memory space, and they are slower to access.

**The typeof Operator**

Since we don't declare a data type when we create a variable in JavaScript, it means we need to somehow have a mechanism to find out what that data type is in the variable at any given time in the application. We can use the typeof operator to do that. What it'll do is return the data type of the passed-in expression, so it returns a string value such as string or number or object or whatever that data type is in that specific variable or expression. So for example, if you do a console.log on the typeof Hello, it prints string. If we do a console.log type 4, it prints number. And this works for expressions as well, so if you have an expression like 4 \* 2, whatever that returns will be then printed out from the typeof. So let's take a look at a demo of using the typeof operator. In this function, typeofSample, I'm declaring several different types of variables. So I've got let product equal to one of the products in the products array. Products array is declared up here. I then have an introDate equals to a new Date. I have an strValue equal to a new String; I have isActive equal to false; I have let result, which will be undefined, and let value equal null. Now, what I'm doing then is, as you can see on lines 50 through 59, I'm doing a series of console.logs to log each one of the outputs from the typeof. So the first one is typeof \_prodcuts. The next one, typeof product, so all the way down the line here. And you can see, even the last one on line 59, typeofSample. That's the actual name of the function. So let's go ahead and try this out and see what each one reports back to us. So here we can see the \_products, that's the array, is an object. When I get just a single product back, that's also an object. Product.productID is a number. Product.productNumber is a string. Strvalue, that's an object. IntroDate is an object. And the reason why is, let's take a look back at those strValue and introDate, if you remember, right here on lines 44 and 45. Those are using the object types, not the primitives. So those are actually objects. So let's go back. So then we see isActive is a Boolean, we see result = undefined, value = object, and typeofSample is a function. So those are the various data types that we have. Like I said, try to use the primitives as much as possible and avoid those object data types.

**Using the Constructor Property to Determine Type**

Let's now talk about object data types and the constructor property. All object data types inherit from Object, not the primitives though, just the objects. So Object has a constructor property, and this constructor property returns a reference to the object itself. It's from this that we can find out a few things about this object. So if we were to look at this little screenshot here, you can see I've got \_products = function Array, product = function Object. These are the constructor properties displayed in the console.log. This is what we're going to see in just a minute in our demo. Object literals and primitives are cast to objects for display, so if we actually say, hey, let's do this in a console.log and take a primitive and apply the constructor property to it, it casts it to an object data type first. Then it can apply the constructor property. So let's take a look at a demo of using our constructor property with various data types. In this constructor sample, again, I'm defining some various values here, and then we're going to go and look at the constructor property. But I want to run this for you first so we can take a look at it here. So this is what we get. We get \_products = function Array, and then some curly braces, and then some native code. So what the constructor property does is, it's kind of an internal representation of this particular object. So an array looks like this, product = function Object. Product.productID is function Number. Now remember, these are primitives, productID and productNumber. So what it does is, it casts those primitives to the appropriate object like number or string. Now introDate was defined as a new Date, so that is an object, comes back as function Date. StrValue is defined as new String, so it comes back as function String. So you get the idea here. So this constructor property, when you apply the toString, it just simply returns a string that tells you what data type it is. Pretty simple. If we look back here at the code, you can see what that looks like. \_products on line 49, .constructor .toString. On line 50, product.constructor .toString. Now, product.productID .constructor, productID is a primitive, but it casts it to the object and then gives it the property constructor so that we can then report that back. And all we get back is a string that we can then output. Very simple.

**Helper Functions for the Constructor Property**

A constructor property is nice, but it simply returns a string. It'd be nice if we had some functions that would return a Boolean to say something like isArray or isDate. Well, we can do that by just creating a couple little helper functions ourselves. Let's take a look at those. So what can we do with this? Well, I've got another function down here called helperFunctionsSample. And what I've done in this particular function is, I've done let introDate = new Date, I've got let result, which is undefined, and let value = null. On line 66 and 67, what I've done is, I'm looking at that array that I had, that \_products array, and I wrote a little helper function called isArray that will return a Boolean, true or false. Same thing for isDate. It returns a true or a false. Then I've got another one on line 70 and 71 that's isNullOrUndefined. So if we take a look down here, we can see these helper functions. Line 75, return value.constructor .toString. Take the constructor property of the value that we pass in, convert it to a string, and then use the indexOf method to see if the keyword Array is in there. And if it is, it'll give us something back which is greater than -1, which means we found it. This expression returns a true. Same thing with the isDate, exact same design pattern. So we could do this for array, for a date, for function, for string. We could add all of these there, and then we'd have these nice, little is methods that give us back a true or false. Same thing for isNullOrUndefined down here. We can return value === null or value === undefined. We now have a Boolean that'll tell us whether something is null or undefined. Again, we can run this. So as you can see here, \_products is Array, true, introDate is Date, true, null or undefined. So this is some nice, little helper functions to have in place for your JavaScript applications.

**The instanceof Operator**

Let's now take a look at the instanceof operator. Now this one is different than the typeof because this one tests if a specific object inherits from the object data type. Remember, everything inherits from object data type except the primitives. So it tests for a specific type of object or for object itself. All right, let's take a look at the instanceof operator. In this function, instanceofSample, I'm going to create a new instance of a product class. So if we take a look down here, I've got this function Product, pass in a few of the properties, set those properties, and that's what I'm doing. I'm creating a new product. Then on line 45, let dt = new Date, let name = new String, let value = A simple string. So I've got a primitive in value. All the rest of these are objects. So now what we can do is, we can test. Look at line 49, prod instanceof Product. So this is going to return a true or false. Prod instanceof object on line 50, so this will return a true or a false. Dt instanceof Date, dt instanceof Object. So we're going to take a look at what each one of these brings back. Let's go ahead and run this, and we'll take a look. Our product function, which is a product class, returns true that it is an instanceof Product, as you'd expect. Let prod is an instanceof Object also returns true because everything inherits from Object. Same thing with dt. It's a new date, so it's using the date object, so it's a date and it's an object. Name instanceof String, absolutely. If we were to do name instanceof Object, since it was declared with new String, yes, it would be an object as well. However, the primitive, which is our variable value, instanceof String, no, it's not. Remember, instanceof only checks for object data types. Primitives are not objects. So those are your different operators there, typeof and instanceof for determining the data type of variables.

**Module Summary**

In this module, we learned that it's important to understand the difference between primitives and objects. There is a difference between those two. And it's very important to use primitives wherever possible. So use those first, and then go to objects if you need them. Remember, objects are slower to access. They take up a little bit more memory. Now we took a look at the typeof and the instanceof operators for helping us detect different data types. We used typeof for checking the type, and we used instanceof for checking what type of object it specifically is. We can also use the constructor property, and this we can use on both objects and primitives. Remember that with the primitives, it can actually cast it into an object, and then we have that constructor property available to us. Coming up in the next module, we're going to talk about the use of the this keyword in different scopes, and we'll talk about the call and apply methods. I hope you'll join me for the next module.

**Understanding 'this' in JavaScript**

**Introduction to 'this'**

This module is Understanding 'this' in JavaScript. For this module, we're going to take a look at what is 'this', we'll learn how 'this' changes based on the global and function scope, based on whether it's in an event handler, within an object literal, called from call or the apply methods, and in constructor functions. Let's get started. The 'this' keyword is often used in many object‑oriented programming languages, and basically 'this' always refers to an object, and that object is typically the one in which the current code is running. Now, we will see that there are of course some exceptions to that rule. Let's take a look at some examples before we dive into some real code. If we're just trying to print out 'this' within a script tag, you have to remember that JavaScript is running within the global window object that's available in every browser. So if you try to just print out this to the console, it's going to come back with, hey, I'm the global window object because you're not within any context other than this global object. Now, if you have an object literal, such as person, and you have a firstName property, a lastName property, and then you have a function called fullName, now when you use 'this', 'this' refers to this object literal called person. If you are using a constructor function such as function Person, now when you use 'this' inside of there, that will depend on which object it is. So look at the very bottom down here. I have let p1 equal to a new person. John and Smith is now assigned to firstName and lastName. So if we were to call the fullName function on p1, the 'this' refers to p1, and that means those variables John and Smith. And then I have let p2 equal to new Person, Bob and Small, so what does that mean? Well, that means that now those two values are assigned, so 'this' is now in the context of p2.

**'this' in Global and Function Scope**

Using "this" keyword will have different values based on in which context your code is executing. For example, when you're in a method inside of an object, 'this' refers to the owner object. In a function or outside of any function, it refers to the global object, or the window object. If you're in an event, it's the element that received the event. Call and apply methods actually refers to the object that's passed in. Now, use strict can also affect 'this', and we'll see that in this first demo. So let's take a look at using global scope, function scope, and how use strict will change the value of 'this'. Here I have a script tag. In lines 15 through 18, I'm out of any function. On line 16, I'm spitting out this.toString to the console. So 'this' will refer to the global, or the window, object. And I'm checking that on line 17 by checking this === window, and that should return a true to us. When you have code outside of any function, then it's going to refer to the global window Object. Lines 22 through 25, I have a function. Inside of this function, I'm also spitting out this.toString to the console.log, and again, I'm checking to see if this === window, and that should return a true. Let's go ahead and run this and take a look. Here is the code from the global scoped code, and you can see that it refers to the object window and this === window is true. Excellent, just like we thought. When I run the function, we also get the global object, and this.window is equal to true. Now, 'this' will change based on whether or not you have use strict in effect, and as I mentioned before, you should always be using this. So let's un‑comment this line. Let's save this, and then let's go back and refresh and click on the function. Now you can see that 'this' is now undefined. Just like what use strict is supposed to do, it says, well, I haven't declared anything called 'this', so it's undefined as of this point. So when you're using 'this' and you're doing just a regular function in the globally scoped namespace, then 'this' does not refer to anything. And that makes sense, and it's actually a good practice to do this.

**'this' in Event Handlers**

For our next example, let's take a look at what happens with 'this' inside of event handlers. On line 9 I have a button and I have onclick event declared inside of there. And I wrote some JavaScript code right inline, and I used 'this' inside of that JavaScript to do this.style.background=read. This will turn the background of this button red. And why does it do that? Because 'this', in the context of this event, refers to the HTML element that it's attached to, in this case, the button. On line 12, I have another button where I do onclick=eventHandler, passing in this. If you look down on line 20, we received 'this' in the variable ctl. So this function gets a reference to the that button HTML element, and then I can go ahead and print that out in the console.log. Let's go ahead and run this and take a look. So inside of the event handler with inline JavaScript we can see 'this' does indeed refer to the button. And if we pass this down, we can see again that we are getting a reference to the HTML button. So 'this' in the context of an event handler always refers to the HTML element to which it's attached.

**'this' in an Object Literal**

Our next example, let's take a look at what happens with 'this' inside of an object, literal. Here's a function called objectliteral. What I'm doing is, I'm creating let product equal to this object. And in this object you can see some properties like productId, name, standardCost, and listPrice. Then we have a function called grossProfit, and inside of here I'm going to access listPrice and standardCost, which are a part of this object, so we use 'this'. So 'this' inside of an objectLiteral always refers to the properties or a method inside of that objectLiteral. And all we're going to do is grab the listPrice, subtract the standardCost, and that becomes the regular price, and I'll spit that out on line 29 to the cosole.log. Let's go ahead and run this, and we can see that indeed it does grab that listPrice and the standardCost, subtracts them, and then applies the currency.

**'this' with call() and apply() Methods**

Let's now take a look at what happens when we use the call and the apply method, and passing in different objects and what happens with 'this'. In this function callAndApply, I have the exact same literal object we just saw, where I create productId, name, standardCost, listPrice, and this grossProfit function. Take a look down here on lines 28 tthrough 31 where I create a new product object called prod2, and I just set a standardCost and a listPrice. Now take a look at line 34 where I do product.grossProfit, so that says the product object, the literal object that we created that has the grossProfit method, let's go ahead and call that, but we're going to call it using product. Then on line 36 we're going to call product.grossProfit.call, but now we're going to pass in prod2. So 'this' on line 34 will be an instance of product. On line 36, 'this' will refer to prod2. So up here on line 21 when we do this.listPrice ‑ this.standardCost, the first one will be the 1431.50 minus the 1059.31 On the second call, it will be the listPrice of 850 the standardCost of 500. The apply works the exact same way. I don't have any parameters to pass, and that's the only difference between call and apply is, apply allows you to pass arguments to that particular method. But other than that, you can still pass in which object you wish to use. Let's go ahead and run this just so we can see what it looks like. And there you can see the two different values based on which object gets passed to the call or to the apply.

**'this' in Constructor Functions**

And let's now take a look at what happens with 'this' within constructor functions. Here I have a constructor function that's pretty much the same as the literal object I created earlier. It's has an ID, a name, a standardCost, a standard price, and a grossProfit function. Now notice the use of 'this' inside of here as well. Well, just like you'd expect, we're also going to try calling this with a couple of different objects. So in our function constructorFunctions, I'm going to create two different instances of our constructor function. The first one I'm assigning to a variable prod1, and the second one I'm assigning to prod2. They have different ID's, different product names, and different standardCost and listPrice. On lines 31 and 32, I call the grossProfit function on each one of those, so prod1.grossProfit, prod2.grossProfit. As you'd expect, 'this' reflects whichever object were working with, so prod1 or prod2. Let's go ahead and run this and take a look at what happens. And as you'd expect, we get the different numbers because we're referencing the different objects, and 'this' then takes on the properties of that object.

**Module Summary**

In this module, we saw how scope determines the value of 'this'. We could get the global window object, we could get the HTML element, or we could get the owner of the method in which we're running. Also, use strict makes 'this' undefined in global functions, so that's something to watch out for. And what is passed to call or the apply methods becomes 'this'. And of course constructor functions owner is 'this'. So lots of different ways for 'this' to show up and have different values in your JavaScript applications. Coming up in the next module, we're going to talk about the spread operator, talk about manipulating arrays, and how we can pass arrays to functions, all using this power spread operator. I hope you'll join me for the next module.

**Using the Powerful Spread Operator**

**Copy a String to an Array Using Spread**

module is Using the Powerful Spread Operator. What we're going to do in this particular module is demonstrate the power of spread. We'll use it to copy and concatenate arrays, we'll pass parameters to constructors, we'll perform shallow copy on objects, and we'll call functions with multiple parameters all using spread. Let's get started. The spread operator expands any iterable object such as a string or an array into another array. It also can be used for passing multiple arguments to a method. The syntax for spread uses the ellipsis symbol, or three dots, it's always on the right-hand side of an equal sign, and I want you to note, currently IE and Edge do not support the spread operator. Let's take a look at our first example of using spread by converting a string to an array. As we learned earlier in this class, a string is an iterable object because it's really just an array of characters. If we have a string, like on line 17, what we can do on line 19 is, we can say let values = …productNumber within square brackets. As you know, a square bracket represents an array. The spread operator iterates over that string and creates a single character out of each value and puts it into the values variable. Let's go ahead and run this. And there you can see the value comes out now into an array. So we took the string and converted it into an array using the spread operator.

**Copy an Array of Primitives Using Spread**

Another use of the spread operator is to copy an array, and let's take a look at how that works. This function copyArray, on line 17 I create an array and assign it the values 1, 2, and 3 into that array. On line 18, I let arr2 = dot, dot, dot, and then the expression. The expression is this variable arr. What that does is, it automatically spreads those elements into the new array. We normally would use the slice method, by the way, to copy an array. On line 21 then, I'm going to show you that this actually, since these are primitive values in the array elements, if we make a change to the duplicated array, nothing changes in the original. So we're going to push a new value onto arr2, we're going to change the first value in arr2, and then we'll spit out both the original array and the duplicated array. So there we see the original array has not changed; the duplicated array, I modified the first element, and I pushed a new value onto the last element of that array.

**Copy an Array of Objects**

When copying arrays with primitive values in it, it's fine, but let's take a look at what happens when we copy arrays with objects in each element. Line 15, I'm saying let \_products equal to an array, and each element of the array now is a literal object, a product object, and you can see I have three elements in here. I then have this function copyObjectArrays. And inside of here on line 45, I let diff = …\_products. So this creates a copy, but since each one is an object, it's actually not copying the objects themselves. It's just copying the pointer to where that object is in memory. On line 48 I'm going to show this by doing diff subscript 0. productID = 999. If we then on lines 50 and 51 spit out the original productID from \_product subscript 0 and diff subscript 0.productID, we see that we changed the new one. We're actually changing the original because their objects, and objects are not copied by value. They're copied by reference.

**Concatenate Two Arrays Together**

Another good use of the spread operator is to concatenate two arrays together. Let's take a look at how this works. I have the same product array that we saw last time, plus I have another array called \_newProducts where I've added a couple of additional ones. And then here is our concatenateArray. So on line 62, let allProducts = \_products.concat, \_newProducts. So this is the original way of doing it in JavaScript, and we can then print out the allProducts.length, and we'll see that it now has five objects in there. The new way to do it with the spread operator, on line 65 is let spProducts = …\_products, …\_newProducts. So what it's doing is, it's taking the original products and expanding it, then it's taking the \_newProducts and expanding that, and again we end up then with a large array. So you see that both of these work, one doing the original concat method in JavaScript, the other using spread.

**Using Spread to Pass Parameters to a Constructor**

We can also use the spread to help us build objects that, where we pass in multiple values to the constructor. Let's take a look at this. In this function, spreadInConstructors, I'm showing you how we create a date. So generally, on line 17, we do let dt = new Date, and you pass in the year, the month, and then the day. So if we spit that out, we'll see 15 Nov 2019. Then on line 20 I'm going to set a dateFields array to 2019, 11, 15. Now what I'm going to do is pass using the spread operator that array to the Date function, and that will now create a new date using these array values. And as we can see, it came out the same. We got two different dates, but two different ways of expressing that constructor.

**Pass Parameters to a Function**

One of the places where spread really shines is when we have function arguments. If you remember the apply method, we would generally pass several parameters to the apply, but we could also use spread in that case and even on your own functions as well. Let's take a look. So look down at line 22. We have this function called multipleParams, and I've got arg1, arg2, and arg3. All I'm doing inside of here is doing a console.log on each one of those arguments. Now, take a look up on function spreadForFunctionArgs on line 15. So we can call multipleParams the normal way on line 16, where we just do multipleParams(1, 2, 3). Now, the other way to do it is to use the spread operator. So if you were to have an array set up with those parameters you need, you can call multipleParams(…args). And that's just the same as line 16. So you can see the exact same thing happens. I like the spread operator because sometimes I do have an array of things that I do want to pass, and it would make it a little bit easier for calling some of my functions.

**Shallow Copy on Object Literals**

Another useful feature of the spread operator is used to perform a shallow copy on object literals. Let's take a look at how this works. In this function objectLiterals, I've created a literal object called product, and I assign it a few different properties. Down here on line 26 I let prod2 equal to, in curly braces, right, that's an object literal, …product. So again, what it's going to do is take each of these objects and perform a shallow copy. Now this is very similar to the JavaScript object.assign method. We then can change one of the newly copied objects, and we could actually change one of those properties and see if it now is a by-reference or by-value type of thing. But since we're doing a shallow copy, I think you already know the answer. On lines 32 and 33 I'll print out each of the objects, and then we'll display the changed value of the productID from both objects on lines 37 and 38. So here we have our two objects. The one did indeed get copied. Only the second one, the copied one, did get changed. So it does actually a copy property by property from one object to the other.

**Module and Course Summary**

In this module, I showed you that the spread operator greatly simplifies your code. It makes your code more concise, can actually make your code a little bit harder to read sometimes though, and it's really not supported by all browsers yet. So you may still have to resort to some of the older JavaScript ways of doing things, but as this gains more and more traction, it will become more used. And I would suggest you be familiar with the spread operator. In this course, we saw how the switch statement can simplify our code, that the for/in and for/of provide us very nice specialized looping, that JavaScript contains a good variety of math, logical, and comparison operators, and short circuiting can make our code more efficient. You now know how to handle exceptions in your JavaScript applications, typeof and instanceof operators provide us a look into our variables, and the 'this' keyword changes based on scope. We also saw that the spread operator, while useful, can make our code a little bit harder to read.