Introduction to JavaScript

Overview

Hi. I'm Liam McLennan. Welcome to an Introduction to JavaScript. In this course I will show you that JavaScript can be a beautiful little language despite what you may have heard. I'll focus on JavaScript the programming language rather than its uses for scripting web pages and AJAX. My favorite thing about JavaScript is its size. It's very small and has a very low concept count. There are a few tricky areas where it behaves differently to other languages you might be more familiar with, but once you understand the differences then JavaScript becomes a comfortable and productive programming tool. Step one will be to take a look at what JavaScript is. Obviously that's something that's important to cover in this course. Once we understand that we'll move onto discussion of why JavaScript is important and why I think every web developer should have a good understanding of the fundamentals of the JavaScript language. After that we'll take a look at the history of JavaScript and try to get an understanding of how JavaScript began, how it got to where it is at the moment, and from there also some consideration of what the future might hold and how JavaScript is evolving now. Next up we will move onto the standard Hello World demonstration. Now, the point of Hello World is to show an example of getting the simplest thing possible up and running in a new language without being distracted by solving any particular problem. Once we've done the basic Hello World, the next thing will be to introduce JS Bin, which is a tool that I'll be using for the rest of this module to show JavaScript examples.

What is JavaScript?

JavaScript is a scripting language. It's very low ceremony. There's no static void main. Generally you can just put some code in a file and run it, and in the case of JavaScript, the file that you put code in is typically a web page, though not always. JavaScript is the language of the browser. It was created as a lightweight language for scripting activity in a web browser, and that's still where it's mostly used. It's a dynamic language. That means that you can modify objects and types on the fly. It's a language with both object-oriented and functional programming features, and that's a statement that's true of most modern languages including things like Ruby, Python, and C#, but in the case of JavaScript it has some very specific support for functional programming built right in that make it a little bit different to work with. It has Java syntax possibly due to the fact that JavaScript appeared around the same time that there was a lot of buzz about Java, but despite that syntax it's not got a lot in common with Java, and the way that the language actually works has more in common with languages like Scheme and Self. The other thing that's important to note about JavaScript, it's very small and very compact and really quite a simple language, which makes it fairly easy to learn. Ubiquity. JavaScript is everywhere. Nearly every computer in the world has a JavaScript interpreter, sometimes several of them. It is possibly the most used language on the planet. There's a lot of research going on at the moment in the area of JavaScript interpreters and a lot of investment from Mozilla, Microsoft, Google, and Apple. JavaScript was almost forgotten for a while and certainly not very interesting to most people until AJAX started to become popular and people started to demand more and more reach activity in the browser, and that really put the focus on JavaScript and trying to take it as far as we can. I said greatness there because JavaScript turns out to be a powerful, flexible, and highly expressive language. There's also lately been a lot of interest in using JavaScript on the server side, and this has come about because of the extremely powerful JavaScript interpreters that have been made available from open source browser projects and also just because of the general increasing interest in the language. I've spoken already about the importance of JavaScript, and I mentioned the JavaScript rich clients. Now I want to demonstrate for you quickly just a couple of the more advanced JavaScript rich clients that are available. What you're looking at here is Google docs, and I've opened up a new web processing document. And you can see it functions very much like the web processes you might be used to like Microsoft would. I can define things like headings and paragraphs. All the normal sort of text features are available, bolding, italics, etc. There is even a Drawing tool. That's not it. So, I can say I want to insert a drawing in my document, select a shape, and draw it. Maybe I'll have an arrow as well, another rectangle. There's a line drawing tool for creating curves, and then I can insert that straight back in my document. Also as you would expect in a standard web processing application I can insert hyperlinks. So, I've just created a link to processingjs, which is another interesting rich client star of JavaScript's website. So, processing JS is a JavaScript library for working with animations and drawings in web pages. The background of the header up here that you can see, it responds to my mouse. This is actually a demo of processing JS animating these circles and lines, and I can influence the animation by clicking and moving around in there. There's also a demo of what you can do with processing in JS. So, this is balldroppings.com. It's a simulated gravity environment with a ball of some description dropping. Now, I'm just going to slow that down a little bit and make it easier to see. And then I can draw on this canvas to affect what happens to the balls. It's a lot of fun. You could spend hours doing this, but you get the idea of---see this is 100% JavaScript and so you get the idea of what's possible in terms of JavaScript rich interfaces. I've mentioned before that JavaScript performance is improving dramatically all the time. You do still have to exercise a little bit of caution. So, in this example you can see it's animating quite a few balls now, and it's very much possible to do this in JavaScript, but you have to be aware that possibly it's not the most efficient way to achieve these kinds of animations. If I just open up my Task Manager you'll see that this is a reasonably powerful machine, but this relatively simple demo animating at most I'd say maybe 1000 balls is pretty much using up all the resources on my machine, and that's in Chrome, which is probably the most efficient JavaScript browser at the moment. In Firefox the performance is quite a bit worse, and in Internet Explorer processing JS is currently not supported. The point I'm trying to make is that JavaScript is powerful and getting more powerful all the time, but there are limits, and you just have to keep that in mind.

History ... and a Demo

JavaScript was born because Netscape wanted to provide scripting capabilities within web pages. Brendan Eich designed the language releasing the first version in the Netscape 2.0 browser. Microsoft soon followed with their implementation called JScript. Legend has it that Microsoft used the name JScript to avoid issues with the JavaScript trademark then owned by Sun. In 1996 Netscape submitted JavaScript to ECMA International who standardized JavaScript as ECMAScript in the ECMA 262 standard. On the Mozilla mailing list, Brendan Eich, the creator of JavaScript, said ECMAScript was always an unwanted trade name that sounds like a skin disease. Fortunately the term is not widely used. JavaScript is not perfect, but for a long time it has been unfairly blamed for flaws in the Document Object Model and differing browser implementations. To quote Douglas Crockford, "The Document Object Model is quite awful, and JavaScript is unfairly blamed. The DOM would be painful to work with in any language. The DOM is poorly specified and inconsistently implemented." The arrival of AJAX and Web 2.0 kick started a renaissance for JavaScript. People were forced to work closely with JavaScript, and in the process they discovered its brilliance. Frameworks like Prototype and jQuery arrived to ease the pain of cross browser development. Today JavaScript is emerging as an important language on the server side with projects such as Node.js, the event-based web server, and CommonJS, the JavaScript standard library. Now it's time to dip a toe in the water and get started with our very first bit of JavaScript code. Here we have the simplest JavaScript project that I could think to write. You can see that it is just a standard HTML page without any content at all. The only thing currently between the body tags is this script tag, which has the type set to text/javascript. That tells the browser that everything inside this script tag is JavaScript code that it should attempt to execute. And what we have here is the window object, which is the top level object within the context of a web page for JavaScript, and on the window object I'm calling the alert function and passing it the string "Hello World." If I grab Chrome and load that page, you see that it shows Hello World in a dialog box. One important thing to note at looking at this example is that browsers pause the page from top to bottom. So, if for some reason the window object wasn't yet available when execution reached this line of JavaScript code then we would get an error. So, it's important to keep that in mind. Often the standard thing to do is define all your JavaScript just before the closing body tag on the page to try to ensure that everything that you need to work with is available. Another strategy is to set up event handlers so your JavaScript code doesn't run until the entire page is finished loading. Now, showing all of my examples by creating these dummy HTML files would get a little bit tedious after a while, so I'm going to use this web service called JS Bin. JS Bin is a simple application that lets you prototype JavaScript code. The screen is divided into two panels on the right hand side. There's the document that we're currently working with, and on the left you see a pane for entering JavaScript code. Now, if I click the Preview button then this page will get executed, and I'll see the output. For each of these tabs you can hide the other one, and quite often I'll work with this view showing just the JavaScript because I'm not going to be working a lot with documents. I just want some way that I can write JavaScript code and execute it. JS Bin will be my weapon of choice for this course, but I want to make sure you're aware of at least one alternative. Interpreted languages usually come with a REPL. REPL stands for Read Eval Print Loop. A REPL is like a programming language shell where you can touch statements and immediately see the results. They're a great way to experiment with a new language. Looking here at the JSDB homepage, and JSDB is a project that provides a simple JavaScript REPL and JavaScript engine for executing scripts. JSDB is based on the Mozilla Spider Monkey Engine. The project is available as a downloadable archive, so there's no install required. I've unzipped JSDB into a directory and changed that directory. Let's have a look at the contents. These are the files that comprise JSDB. This file jsdb.exe is the one we will use to provide a JavaScript REPL and also to execute script files. Running JSDB without any parameters starts the REPL. I can now execute JavaScript code. Let's start with the classic Hello World. JSDB provides a function writeline to print output so that the basic "Hello World" looks just like that, and you can see that it echoed Hello World back to me. Next I'll declare some variables and try some arithmetic, and finally I can exit the REPL with Control+C. If you already have a script file that you want to run, JSDB can do that too. I'll create a new script hello.js and populate it with the familiar "Hello World" example and close that file. And now I can execute that just by passing the file name as the first argument to JSDB, and you can see I've got Hello World printed once again. Finally, it's possible to mix these two techniques, REPL and script files by using JSDB's load function. Load fetches a script file and evaluates it. This means I can start a REPL and load my "hello.js" script. I can see that works as well. It's also possible to load script files from within script files. For example, hello.js could load say an XML processing script or some other resource.

Summary

That brings me to the end of this introduction to JavaScript. The key points I've tried to express are that JavaScript is important. Every web developer needs to understand the basics. JavaScript is significantly different from the mainstream languages that most of us are accustomed to such as C# and Java, but despite these differences JavaScript is easy to learn mostly because it is so small. JavaScript is incredibly powerful. I find myself wishing I could run JavaScript as a first class CLR language. Now that we've covered the background, the next module in this course will examine the building blocks of JavaScript code.

The Building Blocks of JavaScript Programs

Outline

Hi. I'm Liam McLennan. Welcome to the Building Blocks of JavaScript Programs. This module introduces the tools that you will use to construct your programs. These are the brushes with which you will paint your JavaScript masterpiece. First up I'm going to talk about the different kinds of comments that are available in JavaScript programs. Comments are something of a prerequisite when starting with a new program and programming language. You will want to use them to write notes to yourself about the various parts of your application, and it's also helpful to understand the comment syntax to help you read other people's programs, which is a great way to learn. After that I'll take a look at variables. I talked in the previous module a little bit about variables, but here we'll examine the syntax in some more detail. We'll talk about null. It's a special object in JavaScript programs and undefined, which is a similar, yet slightly different primitive type. Sooner or later you will need some assistance, so we'll have a quick look at where we can find help for JavaScript and where the best online documentation can be found. The big subject in this module is objects. We'll go in some detail into the JavaScript notion of an object. And then finally we'll talk about equality, how things compare in JavaScript, and the various things that you need to be aware of there.

Comments and Variables

It's been awhile since I've done any Java programming, but I understand that JavaScript comments work the same as Java comments. The most basic comment available is the single line comment, just a line of text beginning with two forward slash characters. Everything from there until the end of the line is considered a comment. It's also possible to do multiline comments. The starting character is forward slash star. Everything from there until star forward slash is considered to be part of the comment. In the example here I've shown the comment start and end characters on lines by themselves, but it also works if there's comment text or indeed any text on the line with those characters. The final thing to keep in mind is that single line comments can be appended to the end of lines of code, and in this case everything from the double forward slash until the end of the line is considered to be the comment. In a previous module I talked about JavaScript variables and their weakly typed nature. Variables are declared with the var keyword and don't have a type until a value is assigned to the variable, and at that point the type is inferred from the value. If the var keyword is omitted and the variable is declared just by giving it a name and assigning a value, the variable will be declared at the global scope. It effectively becomes a global variable, and this can introduce some subtle and unintended behaviors and also variable name collisions, so good guidance is just to make sure that you always use the var keyword when declaring a variable. Let's take a look at an example. The first thing I'm doing here is declaring a variable using the var keyword, giving it the name streetNumber, and assigning the value 49. Because 49 is a number, the variable type of the streetNumber variable will be the JavaScript number type. In the second example I declare another variable this time called streetName and assign a string, so the type of this variable will be string. I'm just going to use the typeof operator here, which will return a string representing the type of the variable that's passed to it and then just append the value of the variable. I'll do that for both of our variables here and then run it. And you can see that as expected the first variable is a number with the value 49. The second variable is a string with the value Brunswick. I'll show you one more example to demonstrate the unintended global variable problem. A global variable is a variable that's accessible from anywhere within your program. Global variables should be avoided because they introduce coupling between parts of your application that access the shared global variable, but may have nothing else in common. Global variables are difficult to reason about, and they're difficult to debug. In my example I have a function called add that returns the sum of two variables. I'm using an intermediate temporary variable called A, which gets assigned the value of the first parameter. Now, because I've neglected to include the var keyword before A, A will be declared as a global variable so my entire JavaScript application will have access to this variable A and the value that's being assigned here. My first test is to add the numbers 1 and 2 together and return the value. I then immediately print the variable A, which as I mentioned I have access to here because I've neglected the var keyword, so it's been declared as a global, and it's successful everywhere. I then perform another addition, this time 54 + 18, print the value, and then take a look again at the variable A. And what we expect to see is that because A is a shared global variable it will have been updated with the value of 54. So, you can see after each addition A has the value of the first number that's being input to the add function. I've added this if test, which will output "a is undefined" if the global variable A is not found. At the moment because I know that my add function is declaring a global variable A then I don't expect to see A is undefined, and I can confirm that. Now, to fix my add function all I need to do is make sure that I declare the A variable with the var keyword. I also have to remove these lines that output the global variable A because now that A is no longer global that will result in an error. Because the global variable A doesn't exist anymore, I now expect to see the value "a is undefined" output. And there it is.

Null and Undefined

Let's take a moment to look at two special JavaScript values, null and undefined. Firstly null. Null is a JavaScript primitive type that represents the absence of a value. One important thing to note about null is that when it's used in a Boolean expression it evaluates to false, and that gives us an easy way to test that a variable has a value. Let's have a look at an example. I define a new variable, myValue, and give it the value null. Then if I test in a Boolean expression, as I said before null will evaluate to false, and therefore we should see this output "null evaluates to false." I run that now. As expected null does indeed evaluate to false when used as a Boolean expression. The second example I define a variable hasAValue and give it the numeric value 1. Now, obviously 1 is not null, and therefore it evaluates to true in Boolean expressions. Objects other than null and undefined generally evaluate to true in Boolean expressions, but there are exceptions such as the number 0 and the empty string. If we try this second example you see hasAValue has a value, so a not null value generally evaluates to true. Moving on now to undefined. Undefined is also a JavaScript primitive type, and it represents an unknown value, that is a variable or property that has not been assigned. One of the most common reasons to encounter undefined is when a non-existent object property is requested. Just like null, undefined evaluates to false when used in a Boolean expression. Have a look at some examples now. So, you see this first example our variable is declared, notAssigned any value, and then immediately output. If we run that we get the value undefined. As I mentioned, just like null, undefined evaluates as false in Boolean expressions, so we'd expect this output here "undefined evaluates to false," which we can see happens. Okay, this final example for undefined I create a new object, assign it to the variable person. It has one property name, which has the value "Frank." If I then output the property age, person.age, which you can see doesn't exist, because it's requesting an object property that has not been assigned, we expect to see the value undefined. And there it is, non-existent object property: undefined.

Where to Find Help

There will be times when you might become curious about a facet of the language or you might just wish to view some reference material or look up some details about the standard global objects or statement syntax. Maybe you're just stuck. At such times, my favorite JavaScript reference is the Mozilla Developer Network. The URL for the JavaScript information is this one on the screen here. Let's take a look. This is the main portal in the Mozilla Developer Center for JavaScript-based information. The JavaScript guide here is excellent and a thorough guide to the language. It's well worth reading if you want an in-depth understanding. And finally if you are just looking for reference material then the JavaScript Reference is the place to go. I'm working on some code to calculate the circumference of a circle. I have a simple function called calculateCircumference. The input is the diameter of the circle. It multiplies it by 3.14, an approximation for Pi, and then outputs the value. If I run that it works. It tells me the circumference of my circle is 12.56; however, I know from other sources that 12.56 is not an especially accurate result, so I'm going to go have a look in the JavaScript Reference to see if I can get a better outcome. Select the JavaScript Reference. Now I remember seeing a Math library, which is here, and if I have a look in there there's a whole heap of constants, which is great, and one of them is Pi, which could be very helpful for my problem. So, we want to try that. I'll replace my Pi approximation with the standard JavaScript Pi, Math.PI, and that gives me a much better result, but now there are more decimal places than I actually care about, so I wonder if I can format that in a nicer way. Go back to the Mozilla Developer Center and the JavaScript Reference. I think that it's worth taking a look at the Number type to see if there's anything here that might help. There are a whole lot of things on the Number. The toFixed method says it returns a string representing the number in fixed-point notation. That sounds interesting. It turns out that the toFixed method correctly formats a number to the number of decimal places that you specify, so for my little circumference calculator I'd like to see three decimal places. So, when I write the output I can just add toFixed(3). And now when I run that I get a value that's correctly formatted to three decimal places and calculated with JavaScript's Pi constant.

Objects

And so we come to objects. Objects are one of the best things about JavaScript. Everything in JavaScript is an object except for the simple type's string, number, Boolean, null, and undefined. Everything else is an object. JavaScript objects are collections of properties. You might think of this like a hash or hash table. The values of the object properties can be of any type, and there can be as many or as few of them as you like. Objects are declared with the object literal notation, again one of the great features of JavaScript. The object literal notation looks like this example here. As usual we're obviously declaring a variable myObject. The open curly brackets and closed curly brackets define the scope of the object. And here I'm declaring two properties, firstValue with the value of the string A and secondValue with the numeric value 2. There's no class construct in JavaScript, but JavaScript does provide a number of ways for creating new objects from a template and sharing behavior between objects. Let's take a look at some examples. Okay, so the very first example here, the simplest thing you can do with an object is just to declare a new empty object because there's nothing between the curly braces that tells JavaScript that we want an empty object. If we then output the result of calling the toString method on that object, all objects have a toString method, we can see the expected output. So, it's telling us object and the type is Object. Not all that useful so far. To get slightly more complicated, there's an alternative way of creating a new object, is to call the new Object constructor. You might like this technique because it feels a bit more object oriented. Once we have an object, we can go ahead and just directly add properties straight onto an already existing object. So, at the time that it's constructed tim is just an empty object, but then we can just go ahead and add a property as if it were already there, and here I'm giving the name property the value "Tim Bob," which is a string. If we have a look at the output of tim.name we get Tim Bob as expected. The real fun starts when we begin to look at object literals. So, here I'm declaring an object literal and assigning it to the variable john. It has two properties, name and age, with the values "John Blaine" and 21. Outputting these properties John Blaine is the name property, 21 is the age property, and again calling object .toString we get the same output just telling us that it is an object. The built-in JavaScript types come with overloads of the toString methods to try to provide more useful output. So, for example, if we call toString on an array of numbers we get a nice string representation of that array of numbers. Similarly, if we call toString on a regular expression and this is the regular expression literal notation, we get nice friendly output of the regular expression. Again, Boolean objects behave the same way. We get false, a nice string output. And finally, if I call toString on a function the output is the content of that function. You might remember before that I added a new property to an object by directly assigning to that property even though it didn't exist yet, and I can do the same thing with a function. Although the toString function does exist, here I'm going to replace it with a new function that's going to give me better output. So, in this case the function will just append together the properties of my john object. And if we call that, we get a much nicer output from our toString method. Another thing that is important to note when creating objects, there's no limit to the nesting of objects. It's possible to have an object that has a property that contains another object, which in turn may have properties that contain more objects and so on. One quirk that can throw off beginners is that when using the object literal notation that we've previously discussed you have to remember that properties are assigned using this colon syntax, not using equals. If you put equals in there, then this will fail.

Equality

Perhaps all men were created equal, but all objects certainly weren't. In fact, objects are only equal to themselves. All other cases objects are not equal even if they have precisely the same values. On the other hand, primitives are equal if the values match, so in my example there the string "cat" equals the string "cat." It's not the same instance or it's not the same string, but they are equal. There is some confusion to do with equality in JavaScript because JavaScript provides two sets of equality operators. There's the == operator and the === operator, and the difference is type coercion. The first, and you might say default set of equality operators == and !=, attempt to coerce the two operands into the same type before comparing them, and this is generally a very bad thing to do, so the common recommendation is never to use == or != and instead stick with the ==== and !== operators, which will not do the type coercion so it prevents a class of error that occurs when comparing two objects that represent the same value, but have different types. So, for example, if you said one, the number 1 == a string containing just the character 1, the == operator would say that they are equal and the === operator would say that they're not. That might be confusing, so let's look at an example. Okay, first example. I want to show that objects are in fact equal to themselves so I've created an object called joe, which has one property name with the value "Joseph," and then I'm outputting "joe equals joe" and a Boolean test that the joe object does in fact equal itself. When I run that we see that joe equals joe is true. And that's great! Now the other side of this equation was that objects are not equal to other objects even if they have exactly the same values. So, I'll create another object this time calling it joe2 and giving it the same property name with the same value "Joseph." And this time I'm going to output "joe equals joe2" and a Boolean test that the joe object is equal to the joe2 object. I run that, and we can see that joe is not equal to joe2 because the output is false. And the reason that that's happening is that the fact that these two objects have the same contents and are identical. They're not equal because they are in fact separate objects. If I change one of those objects it's not reflected in the other. Primitive types on the other hand, they behave differently. We can compare two separate primitive objects, and if they have the same content, the same value then they're equal. And we can show that here by outputting this Boolean test the string "apple" equals the string "apple," a separate instance of a string, but with the same content. Have a look at that. We see that apple is equal to apple. If the contents are different, "apple" equals "cat," then obviously we would expect that would be false, and it is. The same thing holds for numbers as well. If we check the number 16 is equal to another instance of the number 16 it reports that is true. The inequality operators are just the reverse, so here we're testing that the number 1 is not equal to the number 2. Obviously we expect that to be true, and it is. As mentioned, the == default equality operator is a dangerous thing to use because it coerces the type of the values that you're comparing. So, obviously we know that a number 1, numeric 1 here, is not in fact equal to the string 1 because one is a number and the other is a string. However, if I run this example using the == operator it incorrectly tells me that the number 1 is equal to the string 1. The reason this is bad is it introduces a whole class of bugs including this one here that I've included for an example where I'm testing whether the empty string is equal to the number 0. There's really no way that we could consider that those two values are equal, yet when we compare them with the == operator it will tell us that they are equal.

Functions

Overview

Hi. I'm Liam McLennan. In this module we're going to look at functions. Functions are the core piece of JavaScript. As Douglas Crockford says in this book JavaScript: The Good Parts, "The best thing about JavaScript is its implementation of functions." But functions do behave differently to other languages, so there are some things that must be understood before they can be used effectively. The first thing we'll do is take a look at a function, have a look at what the function looks like and the parts that make up a JavaScript function. Then we'll take a look at the different approaches that can be used when declaring functions. JavaScript provides a number of similar, but slightly different syntaxes for function declaration. Functions wouldn't be much good if we couldn't call them, so we'll have a look at function invocation and how that works. We'll talk about function overloading and how that applies to JavaScript. We'll look at the Arguments Object, which is a variable that's a special variable that's available inside of every JavaScript function. We'll take a brief look at function recursion, functions calling themselves. And finally we'll end with a discussion of closure and how that's implemented by JavaScript functions.

Meet the Function

In the last module I said that everything in JavaScript is an object or one of the JavaScript primitives, and it's still true. Functions are objects. In fact, functions inherit from object. What makes functions special is that they can be invoked. Here we have a very basic function. You already know that we're looking at a function with the name square that accepts one parameter and returns that parameter multiplied by itself. Function's definitions always begin with the function keyword. There's an optional function name, 0 or more parameters to the function, the function body delimited by curly braces, and an optional return expression. There are a number of techniques for declaring functions in JavaScript. The one that we saw in the last slide I call the standard function declaration. Another option is to declare a function with a name and assign it to a variable. If you do this, the function can't be invoked by its name. It has to be invoked via the variable name. So, in this case you would have to call the squareFunc variable. If you're declaring a function and assigning it to a variable, the function name can be omitted. The only reason that sometimes people still include the function name is because then it will appear in the callstack when debugging. The case that we're looking at now, if we were debugging a program that called this function, the function is anonymous, so there would be no name shown in the callstack. Another pattern that you will see quite a bit looking at JavaScript code is anonymous functions that are immediately invoked. So, you begin with a standard function definition and then wrap the function in parentheses, that's just to group it together, and then immediately invoke the function immediately after it's declared. Now, because this function accepts one parameter when it's invoked, one parameter is passed in. The benefits of doing this is that because the function is declared and then immediately invoked and never assigned to a variable, it has no effect on the outer scope, so there's no namespace pollution occurring here. The other benefit is that it provides a technique for aliasing variable names. So, you can see in this example that within the function the variable name is X, but outside of the function it's referred to as numberToSquare. A good example of an anonymous and immediately invoked function is pretty much any jQuery plugin. They're generally implemented using this style for both of the reasons that I mentioned. That is when you reference a jQuery plugin you don't want that plugin to pollute the namespace that's being used by your program, so it's good if the plugin is self contained and it's variables aren't visible outside of that plugin, which is provided by this pattern. Also, within your application you might be using the dollar sign variable for some purpose other than jQuery, so using this pattern allows us to alias jQuery to the dollar sign.

Invocation

Now we can declare functions, but functions aren't much good unless we can call them, and so we come to function invocation. Let's take a look at some example functions. Okay, so here we have the same function declared twice. The first time is the anonymous function assigned to a variable style, and the second one is just the standard function declaration. Now, in the case of the second example here the function is invoked by using the function name followed by parentheses. So, if we were to execute square, pass 7 as a parameter, then we would get a result of 49. The other function declaration, squareFunc, is executed by calling the variable name followed by parentheses. They're very similar. So, if we called squareFunc and pass 7, then again we would get the result 49. This is probably familiar. It's the same way that lambdas are invoked in C#. Take a look at an example. I have here a function called add that accepts two parameters, num and num2, and returns their sum. If we call add 6 and 7 we get the result 13 as expected. What would happen if we passed an extra parameter? So, here I'm passing 6, 7, and 1, three parameters to a function that only accepts two parameters. Let's try that and see what happens. We get the same result. What happens in this situation in JavaScript is that the last parameter, which is not expected, is just ignored. There's no error. I have another function here this time accepting three parameters. So, now what happens when we define the same function twice, firstly with three parameters and then again with two parameters? Now, in C# we would expect that what would happen is that my first call add 6,7 would call the function that accepts two parameters, and the second call add 6,7,1 would call the function that accepts three parameters, but that's not what happens. JavaScript doesn't allow function overloading, so what's happening here because I'm declaring two functions called add in the same scope, the second function just overrides the first one. That brings us to function overloading. As we've already seen, JavaScript functions cannot be overloaded. JavaScript does provide parameter flexibility by matching parameters on a best effort basis. If too many parameters are provided or too few parameters are provided, it will not cause an error. Extra parameters are ignored. Parameters that are not provided are initialized to undefined. When calling a function, object parameters are passed by reference, and primitive type parameters are passed by values. Take a look at an example. Okay, so I have here an add function that we've seen before, two parameters, num and num2, and I'm calling it with three parameters. I'll just quickly run that. You see the result is 3. So, what's happened? It's added the first two parameters together and ignored the third one. If I now call the add method and pass just a single parameter, the result is the special JavaScript value not a number. Let's take a look at the type of the second parameter when it's not supplied, and you can see here that as I mentioned parameters that aren't provided in a function invocation are initialized to undefined. Now to demonstrate that primitive types are passed by value I have a string here called name with the value "Tom" and a function to reverse the characters in that string. If we reverse name and view the output we can see that the output is Tom backwards. If we now have a look at the current value in the name variable, because name is a primitive type it's passed by value. We would expect that the function received a copy of the variable name, and the original variable should have its original value "Tom." And so it does. On the other hand, if we are passing an object into a function, it's passed by reference. So, in this function reversePersonsName we're modifying the object that's passed into the function, so then we will expect that if we later check the original object we'll see that it's been updated with the changes made by the function. First thing though is to reverse the name of the Tom object and check the output. As expected you can see it's Tom backwards. And now if we look at the name property on the original Tom object it's been updated as well because objects are passed by reference.

Arguments Object

The arguments object is a special, magical, local variable available within all functions. It contains the function's parameters, and it behaves like an array in that it's indexable like an array, and it has a length property like an array, but it's actually not an array. It's an object. Jumping into the example now. The first function that I've defined, argumentsType, is here to show that the arguments variable, while it behaves like an array, is actually not an array, so I'm simply returning the typeof the arguments variable, which you can see is an object. When I'm invoking my argumentsType function I'm passing in the value A. Even though the function doesn't have any arguments as we've previously discussed, that doesn't cause an error in JavaScript. So then if inside the function we write out the length property of the argument's variable we're passing in one value, so as expected the length of the argument is 1 just like what you can do with a JavaScript array. If we then were to output the first item of the arguments object, so indexing into it using the square brackets notation, again the same way that JavaScript arrays work, then we should get the value passed into the function, which is A. In my next example I have yet another add function, but this time there are no parameters to the method. It's just going to take some variable number of parameters, add them together, and return the result. So, the first thing I want to try is calling the add function with no parameters at all, and you can see within the add method that we're looping through every item within the arguments object. So, if we pass in no arguments we expect there to be nothing in the arguments object, and therefore it should probably just return 0. And that's what it does. Slightly more interesting example this time, just the one argument 5. So, within the add function it declares a total of 0 and then adds all the arguments. So, if we add 5 to 0 we're expecting an output of 5. And finally, we can pass in any number of integer arguments and it's just going to loop through and add them all together. So, if we pass in 2, 3, 8, and 2 we get 15.

Recursion

JavaScript functions support recursion. Recursion means that a function can call itself. This is often used very effectively to work with tree structures where each successive recursive call deals with a smaller and smaller part of the tree. When defining recursive functions, it's very important to make sure that there is a definite reachable exit within the function to make sure that you don't get stack overflows from recursive methods that call themselves infinitely. Here's an example of a recursive function. It's a factorial calculator. I'm calling the factorial function with the argument 5, and you can see that as part of the return statement the factorial method calls itself with the argument reduced by 1. So, the first time through it will be factorial 5 - 1, so factorial 4. Every time the factorial method invokes itself and is reduced by 1, so eventually we know we're guaranteed to satisfy the guide clause at the start of function that sooner or later N will be equal to 0 or 1, and then the factorial function will finish.

Closure

JavaScript functions are allowed to be nested. It's possible to have a function within a function, and when that happens the inner function has access to variables defined in the outer scope. In fact, the inner function has access to all variables that are in scope when it's declared, and this is the property known as closure. Here's a very common example of a jQuery method call that takes advantage of this closure property. What this code here is doing is performing an AJAX HTTP get request to this URL here. Now, this is an asynchronous operation, so the script is going to start that method call and then continue processing without waiting for the result. And when a result is returned from the server then it will call this callback that's passed as the second argument to the get function. The closure here is the fact that the callback function is able to access the resultSelector variable, which is declared outside of the function and may not be in scope anymore when the callback function is called. But because it was in scope when the function was declared, and because JavaScript has closure, the resultSelector variable remains accessible. Let's jump to an example. This example is a little bit more complicated than anything we've seen so far, so make sure you take the time to really look at it and try to understand it and maybe pause the video if you need to. Remember also that you can always access the jsbin URL here and get access to this code and play around and experiment for yourself. What I've got here are three functions, the outer function here. Inside the outer function there's the middle function, and inside the middle function there's the inner function. Now at each of these levels, firstly at the very top level, I've declared a variable. The first one is A. Then within the outer function there's a variable B, within the middle function a variable C, and right at the end I kick the whole process off by invoking the outer function. When that happens at the end of the outer function it in turn invokes the middle function, and then at the end of the middle function the inner function is invoked. This is demonstrating closure because when the inner function is invoked it outputs the value of each of the variables A, B, and C, all of which are declared in outer scopes. If I run this now we expect to see A, B, C. And that demonstrates closure.

Summary

Well, that brings us to the end of our exploration of functions. We've learned that functions are special objects just like every other object in JavaScript except that they can be invoked. They're declared with the function keyword. They can be named or they can be anonymous. The benefit of naming functions is that when we're trying to debug the name of the function is shown in the callstack. JavaScript functions cannot be overloaded unlike most other programming languages including the .NET programming languages, but they do provide support for an arbitrary number of arguments via the arguments object, which behaves like an array despite not actually being an array. JavaScript functions can be called recursively, that is functions may call themselves. And one of the great functions of JavaScript and the way that functions are implemented is that functions provide closure, that is they capture variables that are accessible at the time that the function is declared.

Control Flow

Outline

Hi. I'm Liam McLennan. This module is about Control Flow in JavaScript. We've talked about objects, we've talked about functions, but to be able to create a program that does something truly useful we need a way to control the flow of execution through our program. That includes being able to branch based on conditions, being able to iterate or repeat certain actions within our program, and having a strategy to handle errors when things go wrong.

Conditional Control Flow

Prerequisite to any discussion of JavaScript's control flow statements, we need to understand blocks. Blocks are a way to group a set of statements together, and it's done by putting the statements between opening and closing curly braces. It's important to understand that blocks do not provide variable scope unlike most other languages that have similar syntax including C#. What this means is that if you declare a variable inside of a block that variable is visible outside of the block as well. The simplest control flow statement is the if statement. To use an if statement you simply provide it with a logical condition. If the logical condition evaluates to true, then the attached block will be executed. If the logical condition evaluates to false, then the attached block will be skipped. The if statement can be extended with an else statement, and the else statement is executed if the if's statement logical condition is false. A more advanced conditional statement is the switch statement. The switch statement provides a multi-way branch based on a conditional instead of just the true/false branch that we get from if else. In JavaScript the switch statement looks identical to a C# switch statement and functions much in the same way. The switch statement works by evaluating an expression and then trying to match the results of that evaluation against one of its cases. When a match is found, execution is passed to that case. When the case is finished executing, it will fall through to the next case unless the break keyword is specified. Here I have a function called introduce with a single argument beatle. Beatle is the expression that I'm giving to the switch statement to evaluate, and then it attempts to match it against labels representing each of the four Beatles. I also have a default case specified, which will be executed if none of the preceding case statements match. Let's take an example. Consider the case where the switch statement here is executed and the value of the beatle variable is Paul. What will happen is the switch statement will first try to match Paul against this first case statement. Since Paul is not equal to Ringo, control will fall through to the next case statement, and the switch statement will try to compare the value of the beatle variable, which is Paul against this case label, which is also Paul. Because that's a match, this write statement will be executed, and "I'm Paul and I play the …aahh… bass" will be output. And then when your control reaches the break statement that will force the switch statement to end. Another example would be this last call to the introduce method where the value of the beatle variable is Elvis. In that case Elvis will be compared against Ringo and then Paul and then George and then John without a successful match being found. Because it hasn't matched any of the cases, control is then passed to the default statement, and this write statement will be called. I'm calling the introduce function once for each of the cases that I've specified and then once for a case that I haven't specified to show the default statement working. When I run this I get the expected result, one line for each of the Beatles plus one line for the value that doesn't have a case statement.

Iteration

Who doesn't love a good loop? Nobody, that's right, and the for statement is the go to guy of the loop world. JavaScript for statements remind me a lot of the Java syntax for a for statement. The for statement consists of three parts after the for keyword. There's the loop initializer, which is run before the for statement; a condition, which is checked every time that the for statement is going to start a new iteration, and if the value is true then the block of the for statement is executed again, if it's false then the for statement is finished; and the incrementer, which is a statement that is run on every iteration. Let's take a look at some example for statements. Just a very basic example this time. I have a simple for statement that initializes a new variable I to the value 1. The condition is that I has to be less than or equal to 10, and every time around the loop we're going to increment I by 1. The body of the for statement simply outputs the value of the variable I. What this for loop is going to do is it's going to execute once for each value of I from 1 to 10, and there we see the output 1 to 10. The for statement is the Swiss Army knife of looping in JavaScript. It's a generic way to iterate. There's also a for in statement, which has a much more specialized job, and that is to iterate over the property keys of an object. It's similar to C#'s for each except that instead of iterating over a collection it can only iterate over the properties of an object. This is best explained by an example. I've declared a very simple object here called tom. It has two properties, name, which has a value "Thomas," which is a string, and age, which is an integer with the value 70. Here I'm using the for in statement to iterate over the property keys of the tom object. Each time through this loop my variable propertyKey will be assigned the next propertyKey from the tom object. So, the first time through the loop the value of propertyKey will be name, and the second time through the loop the value of the propertyKey will be age, and at the moment I'm just writing the propertyKey out. If I run that we get name and age as anticipated. Once I have the property keys for the object I can use this special syntax to access the value of the property having that name. If I run that now I should see the property keys, as well as their values. So, here's my first propertyKey name and its value Thomas, the second propertyKey age and its value 70. We're not done talking about loops. JavaScript has another looping construct, the while statement. The while statement is a pretested loop, which means that the conditional expression is evaluated prior to executing the body of the loop. So, in the example that we see here the expression is keepGoing. Now, if that value is false the body of the loop will never be executed. If it's true, the body of the loop will be executed, and then the test will be applied again. When using these loop statements it's important to make sure that the looping condition will eventually become false. If keepGoing never becomes false then we have an infinite loop. The alternative to while is the do while loop, which is a post-tested loop. This means that the expression is evaluated after the loop body has already been executed. So, in my example here, regardless of the value of keepGoing, the body of the loop will be executed at least once, and then the expression is tested. Once again we want to make sure that the loop condition will eventually become false to prevent an infinite loop. Let's take a look at some examples. I'm setting a variable called keepGoing to true and then using it in a while loop. Because while loop is a pretested loop, if keepGoing were not true then the whole loop would just be skipped, but in this case keepGoing is true so we expect control will enter the loop body. It's going to output while loop. Then it sets the value of keepGoing to false so when the expression is evaluated again keepGoing will be false so control will exit out of the while loop. In the case of the do loop it's post-tested so the body of the loop is going to execute at least once. We'll see this output "do while loop." Then the expression is evaluated. Because it was previously set to false it's going to fail that test, and the body of the loop won't be executed again. Let's run that just to check, and we get the output that we're expecting, while loop and then do while loop.

Error Handling

I'd like to say that nothing every goes wrong, but it would be a lie, so we need some capacity to handle errors when they occur. Luckily JavaScript has a built-in error handling mechanism that you're most likely very familiar with if you have experience with C# or Java. It uses the try/catch/finally model, and we can throw an exception when some unusual error condition occurs. It's best not to use exceptions for expected normal program behavior. Exceptions are thrown using the throw statement. When you use the throw statement it can be given an exception object, which is then attached to that exception. When an exception is thrown, execution stops at that point and control is passed up the call stack until the exception is handled or until the top of the call stack is reached. If the exception is caught, then the exception object can be accessed. This is what it looks like to throw an exception. You can see the use of the throw keyword, and the throw statement is given an object. And here I've given it a name property and a message property, but there aren't any rules about the contents of that object. The power of this style of exception handling is that we can throw an error where it occurs and then choose to handle it wherever it makes the most sense for us. Whenever an exception is thrown within a try block it can be caught with an associated catch block. If the catch block is given an argument, the value of that argument will be set to the exception object that was given to the throw statement. A finally block can also be attached to the try/catch block, and it's used to guarantee execution of a set of statements, even in the event of an exception or error. So, no matter what else happens, we can be guaranteed that the finally block will always be executed. This is the basic syntax for try/catch/finally. We can see that within the try block an exception is being thrown with a name and message property. The catch block has an argument (e), which then allows us to access the values of the exception object. And I've also provided a finally block, which will be executed regardless of whatever else happens. Let's try some examples. This is very similar to the syntax we were just looking at. There's a try block that throws an exception; a catch block, which handles that exception, and then it's outputting the properties name and message of the exception object. If I run that we see the name of the exception SomethingWentWrongError and the message Something went wrong. You should fix it. If I uncomment a bit of code in the finally block we should see that this output occurs, and we see "finally always executes" regardless of the fact that we're throwing and catching an exception perfect.

Summary

We have covered the most important control flow statements in JavaScript. We've looked at the conditional control flow statements if, else, and switch for multiway branching. We worked through the common looping constructs in JavaScript, for, for in, while, and do while. Finally we took a look at how to handle errors in JavaScript and JavaScript implementation of the try/catch/finally style of exception handling.

Types and Libraries

Outline

Hi. I'm Liam McLennan. This module examines JavaScript's Built-in Types and Libraries. Built-in types and libraries are the types, objects, and functions available for you to use in JavaScript. Think if it as JavaScript's version of the base class library. The JavaScript set of built-in types and libraries is relatively small because JavaScript was designed as a browser scripting language. For example, there is no I/O API and no cryptography API, just the basic libraries needed for browser scripting. Anything else you require has to be added by loading additional scripts. This module we will investigate the built-in libraries, as well as some of the more useful open source libraries. Let's start with the string type, obviously a very useful type to have around when dealing with web pages and HTML. And the obvious next thing to look at after strings is JavaScript's number type, and JavaScript does a few things a little bit differently with its numerical data types. Arrays are also very important in JavaScript, one of the main data structures that you get out of the box. And while we're looking at arrays we'll also talk about underscore.js, which is an open source JavaScript library that adds functional programming concepts to JavaScript. JavaScript has great native support for regular expressions, so we'll look at some of the libraries for that. We'll take a tour of the date type. It's somewhat limited, so we'll also have a look at Datejs, which is another open source library. This time its purpose is to make working with JavaScript dates a little bit easier. After that we'll work through some of the library functions that are available within JavaScript focusing on the more common usages, and that includes the math object and the functions that are available for mathematical operations.

String

The JavaScript string is a primitive type representing an ordered set of characters, and the first thing you can notice from the example on this page is that strings are created using one of two literal notations. The first notation, string1, encloses the string in quotation marks, and the second notation, string2, encloses the string in apostrophes. It's useful to have these two notations because it allows you to include the other character in the string. So, you can see in the first example because the string is enclosed in quotation marks I'm able to include an apostrophe in the string without escaping, and in the second example the string includes quotation marks, and I don't have to escape them because the string is delimited by apostrophes. The final thing to note about JavaScript strings is that there is no multiline string syntax unfortunately. In general, JavaScript string escape sequences begin with the backslash character. For example, \n represents a new line. We can escape the string literal delimiters also with the backslash character. So, depending upon the string literal format that's being used, either the quotation mark or the apostrophe may need to be escaped with a backslash, and Unicode characters can be included by using \u and then the code representing the character that you wish to use, all of which can be demonstrated with an example. The first example I have here shows the new line character \n in between two lines, Line one and Line two. Instead of using my write function, I'm using alert to show this string in a dialog box. And the reason I'm doing that is because if I use the write function the new line character will be rendered as a new line in the HTML, and that won't be visible in the output. By using an alert, the new line character will show up as a new line like that, so it split my string into two lines, Line one and Line two. This next example is a string delimited by apostrophes, and I also want to include an apostrophe within the string. Because that's the string literal delimiter that I've used, I have to escape the apostrophe with a backslash. You can see that that's generated the correct output. This next example is a string delimited by quotation marks, and the string includes a quotation, so I have to escape those quotation marks again just by placing a backslash before the quotation. If I run that, we can see that we get the correct output. Finally, this is an example of a string that's including a Unicode character. So, the escape sequence is \u, and then the code of the Unicode character. I'm using 00A9, which is the code for the copyright symbol. So, you can see that outputs the copyright symbol.

String Methods

This should certainly look familiar. To join strings together in JavaScript you use the plus operator, the same as C#. When concatenating strings, it's important to make sure that the things you're concatenating actually are strings; otherwise, the behavior may be unpredictable. There's no string interpolation in JavaScript. String interpolation is a way of baiting strings inside of other strings, and JavaScript doesn't provide a native way to do that. You have to join strings using string concatenation or some other means. There are a number of methods available on strings including charAt, which returns a character at a particular position within the string, and that is 0 based. So, the first character is returned by saying charAt 0. It returns a character, but there is no character type in JavaScript, so it's actually a single character string. The indexOf method returns the starting index of the string that you're looking for. There's a replacement method. We search for the first argument and replace it with the second. The search method is very similar to indexOf except that the argument is a regular expression. The slice method returns a substring of a string. Split is used to split a string on some separator. If no separator is provided or if an empty string is provided as a separator, split will split a string into individual characters. ToLowerCase() obviously makes all the characters lowercase, and toUpperCase() makes all the characters uppercase. Let's try some examples. This first example is using the charAt method to grab the third character in the string "abcd." So, if I run that now I'm expecting to see C because that's the third character, so that's working correctly. You might remember indexOf. It's going to return the index of the search term. So again, searching for C, and we know from the last example that C is the third character, which has the index 2. This next example is using the search method, very similar to indexOf except using a regular expression. So, this regular expression is searching for the character C in the string "abcd." So, again, we expect that is going to be the third character with an index of 2. Now we're going to try some replacements. My starting string here "3 oh 3 it's a magic number," and I'll want to replace the character 3 with the character 4. And you see the output is 4 oh 3 it's a magic number. So, what's happened is the replacement method has replaced the first instance of the character 3 that it found, but not the second. That's the behavior that you will always get when using a form of the replace method that accepts two string arguments. Now, if we take a look at the original lyric variable to see whether it's been modified, we can see that its value is still the original value. So, the replace method returns an updated string, but the original string remains unchanged. Now if we look at a different version of the replace method, this time searching on a regular expression instead of a string, the output is exactly the same. So, it's replaced the first instance of the search term, but not the second. However, if we again use the regular expression form, but this time specify the G flag on the regular expression, G for global, that will tell the replacement that it should replace every instance of the search term, not just the first, and this time we see that indeed it has replaced each instance of the search term. Next is the slice method. Slice is used to extract a substring from a string. Consider for example if I wanted to extract the middle two characters of the string "abcd," so I want to extract the bc part. The first parameter to the slice method is the index of the first character you want to take. So, the index of B is 1, and the second parameter to the slice method is one greater than the index of the last character that you want to take. So, the last character I want to take is C. It has an index of 2, so one greater than 2 is 3. Running that we see that we've extracted bc. Split is a method that converts a string into an array by splitting on some character. So, in my example here I want to split a sentence into its constituent words by splitting on the space character. So, I have my string and I call .split and pass a single space character, and then I'm just looping through and outputting each word. If we run that, we can see the output is each word of the sentence. Finally, toUpperCase starting with the string Frank and converting that toUpperCase we get what we would expect, and then converting the same string toLowerCase each character is converted to lowercase.

Number

All numbers are floating point. You may remember that JavaScript has only one numerical data type, and that is the floating point data type. It's convenient because you don't have to think about the type of the numbers that you're working with, but there's a tradeoff because decimal fractions are not exact. JavaScript provides the standard numerical operators addition, subtraction, multiplication, division, and modulus. The number type also has a handy method toFixed, which returns the number fix to n decimal places. Let's try some examples. So, the first thing is we're going to take a look at the type of the integer value 1. We can see that the type of 1 is number. Next we'll have a look at the type of a decimal value 3.14, and again we see the type is number. So, that is just to demonstrate that there's just the one numerical type, and that is number, which is in fact a floating point data type. Addition works as expected, 1 + 1 we get 2. Subtraction same again, 10 - 1 is 9. Multiplication, three fours are 12. Let's try some division, 22 / 7, we get a decimal number. And modulus, 22 % 10, 2. I mentioned that decimal fractions are inexact when represented with the number data type. to demonstrate that we'll take a look at 0.1 + 0.2. Obviously we're expecting 0.3. And what we actually get is 0.3, then a whole lot of 0's, and then 4. This is where the toFixed method comes in very handy because we can say 0.1 + 0.2 to a fixed number of decimal places, and I just want to see the one decimal place. And that looks more like what I was expecting. The other alternative to avoid this problem of inexact decimal fractions is to work with whole numbers. This is commonly used when working with money. Instead of working with parts of dollars, instead we convert everything into cents. Instead of having 3.5 dollars for example to represent $3.50, we convert that into cents and work with 350 cents. So, my same example repeated again, but this time I convert each of the decimal fractions to a whole number by multiplying by 10, perform the addition, and then at the end divide by 10 to get back to the correct value. And you can see when I do that, because I was working with whole numbers instead of decimal fractions, the result is correct.

Array

You will use arrays in JavaScript a lot because they are one of only two built in data structures that you have. You have the array, and you have the object. Arrays are indexed collections. You can store anything in the array, so the various elements don't have to have the same type. If we look at the example given here, you can see I've created an array. The square brackets are the array literal syntax, and within the array we have a string, a one character string, the number 1, a regular expression, and an empty object literal. When I say that arrays are an indexed collection it means that we can use this square bracket notation to access array elements by their position in the array beginning at 0 for the first element. Arrays also have a length property, which tells you how many items are currently in the array. As I mentioned, the array literal syntax is the square brackets with the items in between, and there's quite a lot of very handy methods on the array object that we will take a look at in this example. The first thing I'm doing here is declaring an empty array and then outputting the length property of an empty array. Obviously we expect the length is going to be 0. Next I have an array of letters, a, b, c, and d, and I'm looping through the array starting at 0 up until whatever is one less than the length of the array, which will be the highest index and just outputting the value that's found there. So, we see a, b, c, d as expected. If I continue working with my letters array that currently contains a, b, c, d, I can use the array's push method to add another item onto the end of the array. So, there's push and pop methods that we can use to make an array behave like a stack. So, if I output that, we can see that E has now been added onto the end of the array. If I output letters.pop, that's going to pop the last element that I just put on the end of the array. It's going to pop it back off, which was E. So, you see we get E back. And now if I check the contents of the letters array, it's back to the starting point because E has been popped off. Arrays have a reverse method that reverses the order of the items. So, again, I'll continue using my a, b, c, d array, but now when I call reverse the contents are now d, c, b, a. Arrays have a slice method that works very much the same as the string slice method. So, the two parameters are the index of the first character to take, and the second parameter is one greater than the last character to take. So, if we slice the array 1, 3 it's going to take B and it's going to take C, which we see here. The splice method is most commonly used to remove part of an array. The arguments are a little bit different to slice in that the splice method the first argument is still the starting index to take, but now the second argument is a count of the number of characters to remove. So, in my example here I want to start removing from the character with the index 1, and I want to take 2 characters. So, the character with the index 1 I'm going to start with B and take two characters, so that will be B and C. So, those two characters will be removed from the original array collection and returned. So, if I run that, we get the two characters that have been removed and the remaining characters in the array A and D. JavaScript arrays have a sort method, which we can use to alphabetically sort the items in an array. I've got here an array with a whole heap of different characters, and I'm just calling the sort method. And the result is the elements of the array sorted alphabetically. If I try again and this time I want to sort an array of numbers, when I call the sort method what we get is not what I had in mind. The numbers are also sorted alphabetically, so they're not in numerical order. You can see it says 1, then 10, then 2. Luckily we can fix this. The sort method takes an optional parameter, which is a function that can be used to sort the items. The function must return 0 if the items are equal, a negative number if the items are currently in the correct sequence, and a positive number if the items being compared need to be swapped. So, for swapping an array of numbers I can implement this comparison function simply by returning the first value minus the second value. And now if I sort using that comparison function, this time the array items are in correct numerical order. I promised there would be additional libraries to help you get more done with JavaScript, and here's one, Underscore. Underscore is an open source library that brings functional programming support to JavaScript. It adds a lot of functions to arrays, and also some functions for working with other functions and for working with objects. Underscore can be used in an object-oriented style or in a functional style, and the decision is really one of personal preference. If we have a look at the example I have here in this page it's a basic demonstration to show the differences in the style. To begin with, I declare an array containing the numbers 1, 2, and 3. Then I'm using the Underscore library and the each function. First let's try the functional style. Here we have the underscore object. I'm calling the each method on that object. The first parameter is an array that I want to enumerate over, and the second parameter is a function with one argument, and that function will be called once for each item in the array. That's the functional style. Now, the object-oriented style works a little bit differently. We use the underscore function to wrap the array that we want to work with, and then we call the each function on that wrapped array. And this time the function has just a single parameter, and again it is a function with one argument that gets called once for each item in the array. So, these two examples, the functional style and the object-oriented style, are exactly equivalent. Before we go onto the examples, let's quickly take a look at the Underscore documentation. It's just a single page, but it is quite a thorough documentation of all the methods. There's a source download of Underscore and a minified and compressed download of Underscore, there's a discussion of the Object-Oriented and Functional Styles that I've mentioned previously, and then there is a list of all of the functions included in Underscore. And if we select any of them, there is some brief documentation and examples. Onto the example. The first examples I have here are the ones that we've already seen in the PowerPoint. I'm using the each function firstly in the functional style and then in the object-oriented style. The each function provides an iterator for collections so that we don't always have to use the for statement, the for loop. This first example is the functional style looping over an array containing the numbers 1, 2, and 3 and just writing the output. So, we see the output 1, 2, 3. Then again doing the same thing, but this time in the object-oriented style. So, first thing I do is use the underscore function to wrap the numbers array, call the each function, and for each item in the array write the output. Again we get 1, 2, 3. The map function is starting to get a bit little more interesting. I'm going to work with the same numbers array containing 1, 2, 3 again. Map translates one collection into another collection, so we provide a function, and that function is called for each item in the original array and returns the value to be placed into the new array. So, in this case I'm using the map function to double each value in my array by multiplying it by two. If we output that, we see 1, 2, 3 doubled is 2, 4, 6. The next function I want to look at is reduce. Reduce takes a collection and distills it down to a single value. Here I'm using reduce to calculate the sum of all items in my doubled array. If we have a look at the output of that, 2 + 4 + 6 reduces to 12. The select function is used to filter a collection. Each item in the original collection is passed to the function we give to select. If that function returns true, then that item is included in the new array. In my example here I'm using a function that will return true if the item is an even number, and this gives me a way to select all of the even numbers from my original array containing the numbers 1 through to 5. If we have a look at the output, we see that it has correctly selected the even numbers. A similar function is the all function. All returns true if the function given to all returns true for each item in the source array. So, this time I'm using a function that will return true if the type of the item is number, and I'm calling it on an array containing two numbers and a string. If we have a look, we see that the output is false because not every item in the array is a number. If I remove the last item of the string and try again, now the output is true. Include is a simple test to see if a collection contains a certain item. I have a list of ingredients, wheat flour, sugar, salt, peanut, can caramel, and I'm going to ask that collection if it includes peanut, which of course it does. When we have a look at the output, we see true. Finally, max is a function that returns the largest value for a collection. This time around I'm using an object as my collection. Most of these functions will work with arrays or objects. Because I'm using an object, max will return the highest value of the properties in my object, which is of course 3.

Regular Expression

Regular expressions are a tool for string pattern matching. They're very common in scripting type languages such as JavaScript, and JavaScript is certainly no exception having regular expressions built into the extent that there is a literal notation for regular expressions. They have three purposes generally. That is regular expressions can be used for searching, replacing, and extracting parts of strings. When you write a regular expression, most characters match themselves. There's also a thing called character classes, which is where a special character or set of characters is used to match certain groups such as backslash lowercase W will match any word character, and \d will match any decimal digit. Square brackets is used to denote a set of characters. So, for example, if we had a, b, c inside of square brackets, then that would match a or b or c. The dot character matches any character, and curly braces are quantifiers. So, for example, here I have .{2} will match any two characters. You can also use a range. So, you could say 2-4, and that would match any two characters, any three characters, or any four characters. Parentheses make a capturing group. What that means is things within the parentheses can be referred to later. The backslash character escapes special characters, and the regular expression literal is forward slash, the regular expression pattern, and another forward slash. I have some examples. Okay, so this first example is simply searching a string "abcde" looking for the character C. The search method on the string type will return the index of the character if it's found. If we run that now, we see that C was found at index 2, which is correct. Okay, now exec is the most comprehensive and powerful method on the regular expression type. If we start with the input this string here "Text with some highlighted parts" and some HTML in there as well, my regular expression is everything between the two forward slashes. So, I'm looking for the HTML element strong containing any character 0 or more times. The star means 0 or more. The parentheses mean that this part of the regular expression between the strong tags is going to be captured so that we can refer to it later, and also notice here the backslash character that is being used to escape the forward slash on the strong closing tag. The next line of code calls the exact method on the regular expression passing my string of text as the argument. And if we then examine the results, the first item in the results array is going to be the substring of my input string that matched the regular expression, which is the strong tags and everything in between. So, you can see it's come out bold because of the strong tags. In the second position in the results array is the first capturing group, remembering the capturing group was the part of the regular expression in parentheses. So, if we have a look at the capture group, we see the text highlighted. It's leaving off the strong tags. Test is a method on the regular expression type that provides a Boolean value indicating if a string matches the expression. In my example here my expression is looking for a decimal digit \d, and I'm testing a string "abc4de." So, because that string does contain a decimal digit, I expect that the output will be true, which it is. The string type provides a replace method, and we can use regular expressions to perform the search part of the replacement, so looking for part of the string to replace. The expression that I have defined here is looking for a word character one or more times, plus means one or more, and that's in a capture group, so we encapture that for use later; then a space character; then again a word character one or more times; and again the second part is going to be captured. The input that I'm going to apply this regular expression to the string "Fred Brooks." So, we can see that a regular expression is looking for two words with a space in between, so it's going to match. Fred will be the first word, and Brooks will be the second word. When using this replace method, the second argument is the replacement to make. Now, that could be a hard-coded string or in this case I've used a function. So, if a function is provided, then the return value of that function will be the replacement. The parameters to that function are the text that matched the expression, the search expression, and then all of the capture groups. In this case, I expect that there will be two capture groups, the first word and the second word, and I'm returning the second word converted toUpperCase, then a comma and a space, and then the first word. Let's have a look at the output. You can see the second word has been converted to uppercase. It's followed by a comma and a space and then the first word.

Date

JavaScript has a Date type, although the Date type has no literal syntax unlike strings, numbers, regular expressions. Dates are declared using the Date constructor, which you can see here. When specifying a date in this format, the example that I'm showing here is actually the 26th of November because months are zero based, so 10 is the 11th month. To get the current date you can just create new Date() without any arguments, and that will give you the current date. Here are some date examples. Firstly just creating a new date for the 26th of November and then outputting that, that's what the toString method on a date object outputs. We can use the toUTCString to get the UTC or coordinated universal time version of the date. It's possible to perform addition and subtraction on dates. In this example I'm creating a new Date() to get the current time. Then I'm calculating the square root of 100,000,000 numbers to try to take a little bit of time. Then I'm getting the current time again and subtracting the first time from the last time to work out how long it took me to calculate the square roots. Have a look at the output of that. We can see that calculating the square root of 100,000,000 numbers took 730 milliseconds. Datejs is a JavaScript library that adds some more functionality to JavaScript dates. The website is datejs.com, which has some nice examples of the kind of things that you can do and a little tester thing where you can for example type in tomorrow, and Datejs is smart enough to workout what you mean by that. Be a little bit careful though. The documentation on the Datejs website is out of date. Here I've provided some examples of the kind of things that Datejs allows you to do. Datejs works by extending the date object. It adds for example a today() method that you can call to get the current day, and then you can do things like add(3).days() using this fluent interface style. You can also get months from the Date object and ask for things like the first Wednesday of the month, again using the fluent interface, and you can also test things like for some particular date you can say is that date a Thursday? Finally, Datejs has a very powerful date parser that allows you to get the date for phrases such as "next tuesday."

eval and JSON

There's a global function called eval that interprets strings of JavaScript code. Eval is slow, has some security problems, can't do anything that can't be done in other ways, and a good idea is just not to use it ever. Having said that, let's take a look at an example just so that we're aware of what it does and how it works. Here I'm using eval to evaluate a string of JavaScript code that's going to display an alert dialog with the text Hello World, and we get the expected Hello World. JSON, also known as JavaScript Object Notation, is a text-based format for representing data. It uses JavaScript object literals as the data format, and it provides a lightweight readable alternative to XML. It's increasingly used in AJAX web applications as a means to transfer data between the client and the server. Here's an example comparing the same data formatted in JSON on the left and XML on the right. Some things to note are that the JSON version is substantially less verbose, and most people consider it more human readable. It's also simply pure JavaScript code, so it's very simple to take some JSON data and deserialize it in JavaScript. To parse JSON, the best way is to use json2, which is a JavaScript library written in JavaScript. It has two important functions, the parse function, which converts JSON to JavaScript objects, and the stringify function, which serialized JavaScript objects to JSON. The alternative way to parse JSON is simply to read it with the eval function, but json2 is preferable because it's faster, it's more robust, and it's more secure than parsing with eval. Let's try some examples. To use json2, obviously the first thing you have to do is load the library into your page. Once you've done that, some JSON code such as this code here where I have defined an object with one property books and the value of that property is an array, the array contains one object, which has a title Frankenstein and an author property Mary Shelley. On the next line I'm using eval to interpret that string of JSON and assign it to a variable called data. On the following line I'm outputting some of the properties of that data object. If we run that, eval successfully passes the JSON string and outputs Frankenstein by Mary Shelley. Next we'll try the same thing using json2. Json2 makes available the JSON object. I'm then calling the parse method and passing it the same input that I gave to the eval function again outputting the title and the author. Json2 provides exactly the same output that we got from eval, but as I mentioned before it's done this faster, more robustly, and more securely. Finally, json2 also provides a way to serialize a JavaScript object back to JSON with the stringify method. Now, I'll take the JavaScript object that I just deserialized and serialize it back to JSON again, and the output is exactly what we started with.

isNaN and parseFloat

Is not a number is a global function that's used as a way to test for numbers and failed mathematical expressions. It's very simple, so I'll jump straight into the examples. My first test here is just to call the is not a number function, pass it not a number. Obviously we're expecting the output will be true. Next I'll call is not a number for the string "hello world." Again, we're expecting true. If we call is not a number and pass it a number, we get false. Last, but not least, if we provide some sort of invalid mathematical operation such as 2 / 'a' and pass that to is not a number, we get true. ParseFloat is the primary means JavaScript provides to convert a string to a number. Like is not a number it's a top level function. There are a couple of examples I've provided here. We can call parseFloat with string a of "48." ParseFloat can also parse more complicated numerical expressions such as this.

Math

The Math Object is really just a collection of functions and constants relating to mathematical operations. There's the abs function, which returns the absolute value of a number; floor, which returns the largest integer less than or equal to the argument; ceil, which returns the smallest integer larger than or equal to the argument; pow returns base to the exponent power; random generates a pseudorandom number between 0 and 1; and round, which rounds to the nearest integer. Okay, so to demonstrate Math.abs firstly I'll try that with a positive number 4.2. The absolute value of 4.2 is 4.2. That's correct. If I call the abs and pass -4.2, we expect the absolute value of -4.2 is also 4.2. Math.floor, remember it returns the largest integer less than or equal to the number. So, if we call Math.floor and pass 1.909, the expected value is 1. Calling floor on a negative number, it's going to behave the same way, so we'll get -2. The inverse of 4 is ceil, so it returns the smallest integer larger than or equal to the number. So, calling ceil for 1.909 should result in positive 2, and calling ceil for -1.909 should result in -1. To calculate 3 to the power of 2, we call Math.pow, first argument 3, second argument 2, and we get the expected value 9. If I call Math.random it's going to return a random number between 0 and 1, so this time 0.84. Obviously if I call it again I expect to get a different value, and I do, 0.42. Math.round returns the nearest integer value. So, if I round 2.5, that should round up to 3, and if I round -2.2 then I expect that to round up to -2.

Summary

This was a longer module than usual. I hope not too long, but I think it's still not too bad considering that I've summarized the vast majority of the JavaScript types and libraries in around about 45 minutes. We looked at the data types that are built in, string, number, date, regular expression, as well as the collection types. There's the object and the array type, both of which you will use an awful lot in your JavaScript programming. If you want to extend object and array and add functional programming support to JavaScript, you can use Underscore.js, the open source library. We also discussed JSON as a format for text-based representations of data and json2 as a JavaScript library for serializing and deserializing JSON, and we finished up talking about some of the built-in functions available in JavaScript, is not a number, parseFloat, and all of the operations on the Math object.

Firebug

Outline

Hi. I'm Liam McLennan. This module will introduce Firebug. Firebug is a developer tool add-on for Firefox and an essential tool in every JavaScript developer's toolkit. Firstly I'll introduce Firebug. We'll open it up, have a look, and a very quick tour of what Firebug is and what it looks like. We'll look at Firebug's capability for inspecting the HTML in a page. We'll look at the console facilities provided by Firebug including console logging, reporting of errors, and executing arbitrary scripts. Firebug provides capabilities for inspecting CSS files that are currently loaded. You can also inspect Network traffic and the currently loaded JavaScripts. One of Firebug's killer features is its JavaScript debugging capability. It's definitely one of the better JavaScript debuggers going around. You can do all the normal things like setting breakpoints, conditional breakpoints, viewing callstacks, and watching variables. Firebug is an add-on for Firefox, and YSlow is an add-on for Firebug. It adds performance analysis features to Firebug.

HTML Tab

I've previously mentioned that Firebug is a Firefox extension that adds developer tools to Firefox. The website for it is getfirebug.com. If you visit the website in Firefox you can install it just by following the install links. There's some documentation and things here as well. Firebug provides tools for working with HTML in a page, working with JavaScript, and that means things like being able to view the JavaScript files that are currently loaded, debugging JavaScript, and working with a JavaScript console. There are similar tools for CSS so you can view which CSS files are currently loaded and what styles are applied to various DOM elements and change those on the fly if you want to. And there are also tools in Firebug for network analysis so you can see the gets, posts, etc. across the wire, view the headers and the bodies of the network traffic. Let's have a quick look at Firebug. This if Firefox 3.5, and I have installed the Firebug add-on. To open Firebug you click the little bug icon in the bottom right of the screen, and the Firebug window opens. You can also right-click and say Open Firebug in a New Window, and that will give you Firebug as an external window. Let's start by looking at the HTML tab. Obviously the HTML tab allows you to view the current markup of the web page you're looking at. It's similar to using View Source except that it will reflect changes to the DOM. So, if you're doing some AJAX and adding or removing parts of the page, the Firebug HTML tab will always show the latest document object model. So, the tab allows you to browse through the document object model. It allows you to correlate DOM elements to the parts of the page, so this goes in both directions. If you select the DOM element in the HTML tab it's highlighted in the page, and you can select a DOM element in the page, and it will be highlighted in the HTML tab. You can also modify DOM elements. This is a cool part of Firebug. Whether you're talking about JavaScript, HTML, CSS, in general it allows you to tweak things on the fly and see what the effect is. I'll go back to Firefox now. And you can see here I'm on the Wikipedia home page. Firebug adds an Inspect Element choice to the context menu. So, you can right-click anywhere on the page, say Inspect Element, and it opens up the HTML tab and highlights the element that you requested. You can also see as I move my mouse over various parts of the page the corresponding parts of the web page are highlighted in high contrast colors. I can browse around inside the HTML view. I can also use the Inspect Element tool. So, you select the Inspect Element tool and then click somewhere on the page, and that part of the DOM is highlighted for me. When something is selected in the HTML tab there's breadcrumbs showing the current nesting within the document. Having selected this anchor tag I can select the div that's inside, and I can selected the outer div again and follow it all the way back up to the body and HTML tags. Also, when a DOM element is selected over at the right hand side here we can see there are a number of secondary tabs. The current one I'm looking at is the Style tab, so it's going to show me all the CSS styles that are applied to the selected element. It also shows the file from which they come, and I can select that and be taken to the style within the CSS file. The other tabs here. Computed shows the browser computed values of all the properties of the selected element. Layout is a very helpful tab. It shows things like the current width and height, paddings, borders, and margins. It's very helpful when trying to work with layouts. As usual for Firebug I can directly select values and change them. And finally the DOM subtab, which shows the detail of the document object model for the selected element. Just as I can edit various CSS values and things I can also directly edit the actual content of the HTML to experiment with some changes and what they might look like. So, for example, I can right-click, select Edit on the div containing this text, and I can edit that and see the results.

Console Tab

The Console tab does lots of things. Obviously it logs error messages, and plus you can write your own console messages to help with debugging. It shows errors when they occur in JavaScript. It shows AJAX network traffic. That's not all network traffic. Loading images and scripts and things isn't shown in the Console. It provides a JavaScript console that you can type expressions and work with the JavaScript in the page, plus it has JavaScript profiling. So, you can profile your code doing different scenarios and see which functions are taking the time. Let's have a look. So, I open Firebug, switch to the Console tab. My first example here is calling a method console.log. When that runs it will write the message to the Firebug console or at least it would except that jsbin is intercepting the console log messages and displaying them in the page. If I deliberately create an error you can see that the JavaScript error has been thrown, and it's logged in the Firebug console with details of what went wrong and a link to the location of the error. This bit of code is using jQuery to perform an AJAX get request. AJAX get requests appear in the Firebug console. You can see it perform the request there. There's all sorts of information included here including how long it took, 641 ms; the HTTP return code, 200 OK. It's telling me that it was a GET request to the URL jsbin.com. I can have a look at the Response Headers returned from the server, as well as the Request Headers, the raw response and the HTML rendering of the response. Since this is a JavaScript console, I can also directly type in JavaScript code here and execute it on the page. So, it's using a bit of jQuery here. That bit of code is going to select a div in the page with an ID of control, which I happen to know exists, and I'm going to return the text value of that div. I can also use this to make changes in the page. So, I appended some HTML to the control div, and you can see that it's updated the page. I can also call standard library functions such as Math.random. Now let's take a look at the JavaScript profiling. I'm opening up a website that I know has some reasonably advanced JavaScript. Behind the scenes there's JavaScript used here to update this graphic when it's clicked. I've got Firebug open, and I'm on the Console tab. If I now click the Profile button to start profiling I now interact with the page by clicking it executing some JavaScript. When I'm finished profiling just click the profile again to stop it, and now I get the timings of everything that's happened while the profile has been running. I can inspect this output and see which methods are taking the most time and where the best possibility of optimization lies.

CSS Tab

The CSS tab shows us what CSS files are currently loaded, and we can view the contents of those stylesheets and interact with them. So, stylesheets can be edited, and the changes to the page are visible in real time. For an example, go to look at the main Google search page. We'll open up Firebug and then switch over to the CSS tab. There's a dropdown list that shows all of the stylesheets currently loaded. When you select one of the stylesheets its contents are shown inside the CSS tab window, and changes can be made directly. For example, if I want to edit the margin of the page I can set this to 100 px and immediately see the effect on the page.

Net Tab

The Net tab shows network activity grouped into categories by All or HTML, CSS, JavaScript, XHR or XML HTTP requests that shows the AJAX requests, Images, and Flash. If for example we take a look at a popular website such as the Australian Yahoo 7 site, if we open Firebug, switch to the Net tab, and then reload the page, if I select the All filter this shows every request that's been made to display this home page. If I scroll right down to the bottom you can see there was a total of 112 requests required to render the page, a total of 822.7 KB of which 745 came out of the cache, and the total page rendering took 8 seconds. If we look at each one of these requests, we can see the individual timing information. So, this first request for au.yahoo.com took 344 ms. When we mouse over we can see that broken down into the various request phases, and in this case there's 195 ms of waiting and the 149 ms of receiving. For each request it tells us the HTTP verb, so this was a GET request; the URL that was requested; the HTTP Status code of the response; the Domain it came from; and the total amount of data transferred. If I select the HTML filter, that will show me the requests just for HTML pages. So, this first one is the original request from my browser. As you would have come to expect, we can see the Request Headers and the Response Headers, the raw HTML response, and the HTML response rendered. The CSS filter shows all of the stylesheets that were loaded for the page. Again, we get Request and Response Header information, the raw response, and you can see that this CSS file has been minified. That means that all the comments and whitespace have been removed to limit the file size as much as possible. And there's also information about the Cache. The JavaScript filter is very similar to the CSS, same information, the Headers, a minified Response, and Cache data. So, in the case of this Yahoo 7 website there were 16 different requests required to serve the JavaScript. The XHR requests show the AJAX requests. In the case of this particular website there's only one. It's passing these three parameters to an AJAX service, and then the Response is a chunk of HTML. Again, we have Cache information. The Images filter shows all of the images loaded on the page. There are 79 in this particular case. If we mouse over we get a preview of what the image looks like and all of the usual information, the domain and the file size and the timing information. Finally the Flash filter shows the flash movies that are loaded, and for each flash movie we can view the Headers and the Cache information.

Script Tab

Of course this course is about JavaScript, so you might say the Script tab is why we're really here. It allows you to inspect the JavaScript files that are loaded and then also inline scripts. You can view them, you can set breakpoints, and you can use the console window to interact with them. From a debugging point of view, we can set breakpoints including conditional breakpoints, we can look at the callstack, and we can set Watch expressions. Let's try an example debugging. Here we have a website with a little bit of JavaScript in it. I'll open up Firebug and jump to the Script tab. If we take a look at the scripts that are loaded, it says to reload to see all sources, so I'll do that. Okay, so we can see the script files that are loaded now. There is a version of jQuery 1.3.2; there's the jQuery UI library; there are some scripts in the actual HTML page inline, which show up as the domain name; there are three jQuery plug-ins, the Google Analytics script; and a JavaScript file called shirt.js. If we have a look at that one, we see a very simple JavaScript file, two functions. When the page is loaded, then this putOnShirt function is going to be called. So, if I want to set a breakpoint there I can just click over in the left margin, and now when the page loads, when it gets to this point, execution will stop as for a normal debugger, which I can demonstrate just by reloading. Execution has stopped at this breakpoint. We can see where we are. It's highlighted that line. The Watch window shows the variables that are currently in scope. I can look at the callstack and also a list of breakpoints, which is currently just one. I can Step Into, Step Over, or Step Out or Continue execution, so stop debugging. If I choose Step Into it moves to the putOnShirt method, and from there I will just let it continue. Sometimes you want to set up breakpoints, but only have the breakpoint stop under certain conditions. So, to achieve that we can set a conditional breakpoint. You do that just by putting a null breakpoint and then right-clicking on it. So, in this case I would like to have a breakpoint that only breaks if the value of this query string variable slogan is not an empty string, so I can do that just by putting a little bit of JavaScript code in here. So, it's going to read the query string value slogan, and it will stop here if that value is not equal to the empty string. I'll clear the other breakpoint. Now, I can reload the page, and the breakpoint wasn't hit because there's no query string variable called slogan. Now, if I type something into this text box and submit the form it's gone and set the slogan query string variable, so now my conditional breakpoint is hit. I can jump over to the Console tab and use that same jQuery function to read the slogan variable, which we can see is base64 encoded, so it's not terribly helpful in this particular case. The callstack now shows that I'm inside the putOnShirt method, and there's a heap of jQuery or anonymous functions higher in the callstack. Another thing that I can do is add new Watch expressions, so this is keeping an eye on variables. I want to notice if they've changed. So, from my example I want to watch the window.location objects so I can add a new Watch expression like that, and you can see that it will show me the window.location value as it changes over time.

YSlow

YSlow is a performance analysis plug-in for Firebug. It adds itself as a new tab to the Firebug window, and when you tell it to, it analyzes a web page against a set of rules and provides recommendations to improve performance. The set of rules comes from Yahoo's Best Practices for Speeding Up Your Web Site, which is in itself an excellent resource. It's a big list of all of the things that you need to do to make your website fast. And when you run the YSlow tool, it will point you to specific rules where it thinks you can make an improvement. Let's try YSlow out by running it against the OnAShirt site that we just looked at. I've already installed the YSlow plug-in, and as usual it's opened by clicking on the icon in the bottom right hand corner of the screen. There are a number of different rule sets here. I'm going to go with the YSlow(V2) rule set. And now all if have to do is click the Run Test button. Okay, so that's finished already. It's given me an overall grade of C and performance score of 79. It's made a whole heap of recommendations broken down by categories. So, in the Content category I'm looking pretty good. The only area where it suggests improvements is on making fewer HTTP calls. It says that I'm loading 7 external JavaScripts, which is a bit much, so maybe I should combine them into one. If I click the Read More link to get some more information about that recommendation, it takes me back to those Yahoo performance rules. I get an A for Cookies. For CSS I've got a C for Minify JavaScript and CSS. So, it wants me to minify my resources, removing the whitespace, and shortening them as much as possible. Images is fine. Again, with JavaScript it wants me to minify, and there are quite a few recommendations here under the Server category, use a Content Delivery Network, Add Expiration headers, Compress contents with gzip, and Configure entity tags. And if you don't know what any of those mean of course you can just follow the Read More link and get the detailed information from Yahoo's website.

Summary

Firebug is an awesome tool for web developers and people working with JavaScript. I often hear people say that the reason they use Firefox is because of Firebug. It's got the HTML tab for examining the current state of the DOM in a web page. There's a Console tab where we can execute arbitrary bits of JavaScript, view errors, and write log messages to help with debugging. The CSS tab shows us the loaded stylesheets and allows us to edit them. The Net tab, which pretty much shows everything that's happening on the network, so the initial load of all the resources, as well as any AJAX requests that are made. There's quite a bit of detail in there. You'll remember we can view the request and response headers, caching information, and performance timing data. The Script tab is where we get to really play with JavaScript. We can see which script files are currently loaded, view the inline scripts in the page, set breakpoints, debug, view callstack, and use variable watches. And the very last thing we covered was Yahoo's YSlow plug-in, which is a nifty little plug-in that makes recommendations on how you can make your website load faster.

Testing JavaScript

Outline

Hi. I'm Liam McLennan. Welcome to Testing JavaScript. This module will look at the different reasons why you might want to test your JavaScript and the tools and strategies available to help you with that. The first topic we'll look at is testing. We'll talk about why it's important and what kind of testing you might want to do for your JavaScript applications. We'll talk about QUnit, which is a tool that's part of the jQuery project. It makes it really easy to write JavaScript unit tests. An important topic for web developers is how to write tests that test DOM interactions. So, we're talking about testing JavaScript that interacts with web pages. I'll briefly introduce Behavior Driven Development, something that's become very popular in the last few years, and it applies as much to JavaScript as to every other platform. Specifically, we'll look at how to do Behavior Driven Development tests with a framework called Jasmine. And finally we'll look at ways that we can integrate our JavaScript testing into an automated build process.

Why Test?

When I talk about testing I'm talking about coded tests that execute automatically. We can write our tests that verify all the behavior in our application matches the behavior we expect and somewhere click a button and get a report stating which tests pass and which tests fail. And of course the most important statistic we want to know is are there any failures at all? Is there something going wrong in our application? The motivation for this kind of testing is to prevent runtime and logical errors. So, why would we want to write automated tests? Well, JavaScript is dynamic, weakly typed, and interpreted. This provides power and flexibility at the expense of compile time checking. Having tests in place is the only safety net that's available to protect the programmer from himself. Because JavaScript is interpreted, we need tests in place to catch the bugs that a compiler would normally pick up when the software is built. Also, without having tests in place it can become very scary to make a change because there's no way to verify that the change hasn't broken anything. We build a suite of tests to facilitate confident refactoring, that is making changes to our code and being confident that we haven't broken anything, which is another way to say that we want to be able to catch regression bugs. The regression bugs are bugs introduced into old software by new changes.

QUnit

QUnit is a unit testing tool for JavaScript. I'm guessing they called it QUnit because JUnit had already been taken. It's powerful and easy to use. The documentation is on docs.jquery.com/Qunit. It's there because QUnit is part of the jQuery project. In fact, it was created as a tool to test the jQuery library. When you use QUnit you define and run the JavaScript tests in a web page. Tests are organized into modules, and modules are declared by calling the module function and passing a name. Within a module tests are declared by calling the test function. Give the test a name and provide a callback to be executed for that test. Within the test you have a standard set of assertions available. The ok assertion is used to test a Boolean condition, equal is a shallow comparison to test that two objects are equal, and deepEqual is a deep recursive comparison of two objects. Every QUnit test page starts with this template. The important things to notice are the inclusion of the QUnit stylesheet, the inclusion of the QUnit script file, and the standard HTML content that's used to display the test results. Let's have a look at an example QUnit test. Here you can see the QUnit stylesheet that I talked about. Here's the inclusion of the QUnit JavaScript file and the QUnit standard HTML. Then below that I've defined a script block, and within the script block is where we write our tests. So, the first thing the declaration of a module, so I've created a new module called "ok assertion," and in there I'm going to demonstrate the ok assertion. All the tests from that module definition until the next module definition are going to be considered part of that module. So, here's my first test definition. I've defined a new test called "passing test" and supplied a callback function. Within my test function I'm using the ok assertion and giving it a true value, which will cause the test to pass and provide a message. The next test is the opposite. The test is called "failure." Again I've provided a callback function, and I'm calling the ok assertion, but this time I'm giving it a false value, which is going to cause this test to fail, and the message will be shown. After that I have declared a second module called "equal assertion," and the tests in this module demonstrate the use of the equal assertion. The first test is called "equal pass." The two parameters are the actual value and the expected value. You can see in this example I'm comparing 1 + 1 to 2, so obviously we expect that test is going to pass, but then in the next test "equal fail" I'm comparing 1 + 1 to 7. So, obviously since those values aren't equal we're going to see a failure for that test. The third and last module is demonstrating the use of deepEqual assertion. I have two tests. The first one, which we'll expect to pass, is comparing two arrays containing the same values. The second test is also comparing two arrays, but this time the arrays have different values. The first one contains 1 and 2, and the second array contains 1 and 4. All I have to do to run these tests now is open this HTML page in a browser. QUnit will pick up all the tests that I've defined, execute them, and report the results. So, here we see all of the tests I've defined. The failing tests are highlighted in this pink color. This is showing me all of my tests on the same page. If I wanted to view just one test at a time I can double- click on it. So, I'll double-click on this first test, and I can expand the detail. So, what we can see here is the name of the module, the name of the test. These numbers here represent the number of failing assertions, the number of passing assertions, and the total number of assertions. So, in the case of this first test called passing test I have 0 failing assertions, 1 successful assertion, and 1 assertion in total. Here's displaying my test message, and at the bottom of the test suite we get overall statistics on the number of tests passed and failed and total, as well as the total execution time. Looking just at the second test now, so this was my failure example calling ok false, very similar to the last one. As I mentioned, it's highlighted in pink to show that it's a failure. We see one failed assertion, 0 passing assertions, 1 assertion in total. The third test was testing that 1 + 1 is equal to 2, which we can see is successful. The fourth test was that 1 + 1 is equal to 7. We can see that the test has failed, and we get a nicely formatted error message showing us that the Expected value was 7, the actual value was 2, and we get a Diff comparing the actual and expected values. There's also a reference to the line number where the error occurred. The fifth test was demonstrating a passing deepEquality test. I compared two arrays containing the same numbers, and we get a successful test there. The sixth and final test, again testing the deepEqual assertion, but in this case I was comparing two arrays containing different values. And again we see an output showing the Expected values, the actual values, and a Diff of the expected versus actual values. The one last feature of the QUnit output is up at the top of the page. There's a checkbox marked to hide passed test. You can use that if you're only concerned with the failing tests.

Testing DOM Interaction

Because QUnit runs all the tests in the context of a page, it's particularly well suited to testing JavaScript that works with the Document Object Model, maybe modifying it or reading some values from it. The test can manipulate the DOM and then assert that the expected changes happened. In your QUnit test page you can set up whatever HTML is required for your test. When doing these kinds of tests, jQuery is a big help to do the DOM manipulation. When doing these kinds of tests, jQuery is a big help with the DOM manipulation and inspection. I have a demo to show what I mean. It's another QUnit test page. Again we have the standard CSS file, script file, and the standard QUnit markup, but this time I also have some special test markup that my tests are going to work with. They're going to manipulate it. I've grouped the whole thing together in a div with the ID "test area." Inside that div I have a text area with the ID "phrase," a "button" with the label "paste," and a div with the ID "target." My test is going to enter some text into the text box and then click the button and then assert that the contents of the text box are written into the div with the ID "target." Once again I've added a script to the page, I've defined a new module called "paste feature," and written a test called "paste adds to target." Inside the test I've broken it up into three phases. There's a given phase where I arrange the context for this test setting up the phrase that's going to be placed into the text box. This bit of code here is a jQuery function call that selects the DOM element with the ID of phrase and then sets the content of the text box to the variable phrase. Next I'm getting a reference to the div with the ID of target, and I'm storing the current value of that div for later use. In the when phase of the test I'm using jQuery to get a reference to the button, and then I'm triggering a click event on that button. Finally, in the then phase of the test where we want to assert that the expected behavior matches the actual behavior I'm using the equal assertion to compare the text of the target div to the combination of what was in the target div plus the text that I've just added. The code for this feature is defined in an external JavaScript file, which I've included, and it's called phrasePaste. If we quickly have a look at that what you see here is a self invoked-function called handleClick, which I'm calling immediately after it's defined and passing a jQuery object in, which is then aliased to the dollar sign. This code gets a reference to the button, and it configures a click event handler for the button. When the button is clicked it gets a reference to the target div, gets a reference to the input, and appends the value of the text box to the target div. Let's run that test now and see what happens. Again, all if have to do to run the test is open the HTML file with my browser. Here you can see the test markup that was added to the page, the text box, the button, and the target div. The test phrase has been written into the text box, the button has been clicked, and the contents of the text box have been appended to the target div. Consequently, the test has passed.

Behavior-Driven Development with Jasmine

Behavior-Driven Development is an outside-in style of development starting with tests that specify the behavior of the application. The tests are known as specifications because they specify behavior, and they're written from the point of view of the end user. The scenarios are often specified using the given/when/then syntax, which is a nice neat way to divide tests logically into the context in which the test is being run, the action that occurs, and then a set of observations where we assert that the expected behavior is the actual behavior. In the JavaScript ecosystem there's a tool called Jasmine for Behavior-Driven Development. The syntax is reminiscent of a similar tool for Ruby called RSpec. The documentation for Jasmine and the download is online at pivotal.github.com/jasmine/. Here you'll find examples, documentation, and all of the download links. The BDD specifications are grouped into suites, and suites are declared using the describe function. Inside a suite context can be established for all the specifications using the beforeEach method. Each specification is then declared using the it method. The it method takes a string describing how the application should behave and a function. Inside the function we perform some action and then assert the expected behavior. Let's have a look at an example Jasmine test project. This is going to look rather similar to the QUnit test. Again, Jasmine tests are based around an HTML page. Also like QUnit there's some standard files we have to include, a CSS file, and in this case a couple of JavaScript files. We then include the source files that we want to test, and we include the specifications. And finally, once all those scripts are loaded, there are a couple of JavaScript calls to start the testing process. In this example the specifications are stored in a file called CalculatorSpec.js. As I mentioned, the specs are grouped into suites, and suites are declared with the describe function. So, we have a suite called "Calculator." Within that suite there's a variable also called calculator, and there's a beforeEach function, which initializes that calculator with a new Calculator object. It's possible to nest suites, so here within the Calculator suite there's a new suite called "when adding positive numbers." So, that's going to group all of the specifications that have to do with the adding of positive numbers by a calculator. Before each of those of those specifications, two variables are initialized, firstNum and secondNum, and given positive integer values. And in this suite we have just a single specification saying that the calculator "should calculate the correct results" when adding positive numbers. The calculator object is used to add the two numbers together, and the result is stored in a variable called result. We then use some of the assertions built into Jasmine to specify that we expect the result toEqual 68. This file includes another suite of specifications, this time for adding positive and negative numbers. Again the beforeEach method is used to set up the context before running the specifications, and this time we're initializing three integers, two of which are positive, and one of which is negative. There's then another specification saying that it "should calculate the correct result," again calling the add method on the calculator object, this time adding together three numbers and expecting that the result will be equal to 700. If we now open the file SpecRunner.html in a browser, we get a rather similar output to QUnit except that it's telling me that one of my specs is failing. When adding positive and negative numbers, it should calculate the correct result. I'm expecting a result of 701, but the actual result is 700. If I go back to my specifications and take a look at the context that I've set up, 14 plus 54 plus -769, I can see that it's actually my specification that's incorrect because when those three numbers are added together the result is in fact -701. So, I correct that error, swap back to the web page, run it again, and this time we see two successful specifications.

Adding JavaScript Testing to Builds

Assuming you have a nice automated build process in place to compile your application, run the tests against it, and maybe even do some deployment, then Jasmine has some good support for integrating into that build process. There's a command line entry point that uses Selenium to launch a browser, execute the Jasmine specifications, and report the results back to your build process. To use the command line entry point for Jasmine, firstly you need Ruby. Ruby is a programming language, another interpreted language like JavaScript, and you can get Ruby for any platform from the ruby-lang website. Just download and install it. Once you have got Ruby, you can install Jasmine using Gem. Gem is a package manager for Ruby roughly equivalent to the NuGet project from Microsoft. So, to install Jasmine you need to run gem install jasmine. This will install Jasmine and all its dependencies. Now that I've installed Jasmine, I can create a new test project using the jasmine init command. That's created two directories, one called public containing my JavaScript files that I want to test, one called spec containing my tests or specifications, and a rake file, which is the Ruby equivalent of a batch file. Firstly I want to show a failing test, so I'm going to open up the spec and introduce an error. The demo specification file that's installed is called PlayerSpec. It has a number of prewritten specifications. I want to see a failure, so I'm going to introduce a problem by saying that I expect true toEqual false. And now I can run that test with the command rake jasmine:ci. This is going to use Selenium, the browser automation tool, to launch a web browser, run the specifications, and report the result. And you can see there it found 5 examples and 1 failure. Now I'll go back to the specification and remove the error that I introduced, and now we expect to see a passing test. So, I run it one more time, and all our specifications pass.

Summary

So, that's the end of testing JavaScript. It wasn't meant to be a thorough step-by-step guide to every possible way to test JavaScript code, but more of an introduction to at least get you started and show that it is possible and a desirable thing to do. JavaScript certainly requires more thorough testing than statically typed compiled code that you might be used to working with in the C# and Java's of this world, but there is good tool support in place. QUnit is an excellent tool. It's great for unit testing and for testing DOM interactions. If you're interested in Behavior-Driven Development, that's available for JavaScript as well. There's the Jasmine tool, which is a lot like QUnit, but it has a Behavior-Driven Development API, and it also has support for integrating with automated build processes.