

GYMNASIUM

JAVASCRIPT FOUNDATIONS

Lesson 1 Transcript

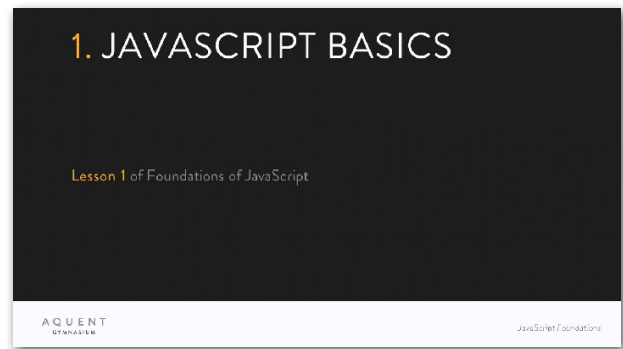
JavaScript Basics

ABOUT THIS DOCUMENT

This handout is an edited transcript of the JavaScript Foundations lecture videos. There's nothing in this handout that isn't also in the videos, and vice versa. Some students work better with written material than by watching videos alone, so we're offering this handout to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one. We encourage you to use this handout alongside the videos, rather than as a replacement of them.

INTRODUCTION

This is JavaScript Foundations, an online course developed by Aquent. JavaScript Foundations, or ‘Stop Faking It, and Unleash the True Power of JavaScript’. This is lesson one, JavaScript basics. In this lesson, I’m going to cover the bare basics of what you need to know about JavaScript. In future lessons, I’ll build on that knowledge to give you a solid foundation in the language. At the end of this lesson, there will be an assignment, and a quiz, so pay attention.



My name is Keith Peters. I’ve been coding since the mid-80s, and I’ve been a professional developer for the last 15 years, or so. I’ve also written, or contributed to, about a dozen programming books, spoken at many tech conferences, and have done many other projects relating to programming education. I love teaching others about coding, and I think we’re going to have a fun time together on this course. But enough about me. Let’s talk about you.

I’m going to make some basic assumptions here about who you are, and what you know, just to make sure that we’re on the same page, starting out here. My first big assumption is that you’re familiar with some basic web technology. You know some HTML. You know some CSS. You know how to create at least a basic website and get it onto a server, somewhere on the web.

I’m also assuming that JavaScript, itself, is not an entirely foreign beast to you. If you’ve been working on the web, you’ve probably encountered or even created the occasional script tag; maybe coded a button click to navigate to a new page, or something simple like that. The idea of this course is to take that basic knowledge to the next level, where you can actually feel comfortable coding more complex behaviors in JavaScript, and maybe even program a simple application.

Some of you may have used JavaScript more extensively in the distant past. And if so, you might not be aware that it’s a whole new world out there, and what JavaScript can do, and how it’s used. If that’s you, this course will bring you up to date. And finally, I’m going to assume that some of you may have a bit of experience using jQuery, or some similar, do-everything, JavaScript library.

A few years ago, something like jQuery was an absolute necessity. Browsers had such massive and basic differences in how they implemented basic features that without a catch-all library like jQuery to standardize everything, your code would wind up being a mess of browser checks. Things have improved drastically over the last few years, though, and jQuery isn’t the vital necessity that it once was. So, if you’re one of those jQuery junkies, this course will help you break the habit, and see how simple it is to implement a lot of the functionality you’d usually rely on jQuery for. I’m not saying that you have to, or you should stop using jQuery. But, at the very worst, you’ll have a better understanding of what’s going on behind the scenes, if you do use it.

So, what am I going to cover in this course? Well, as I've said, in this lesson, I'll start to show you the tools you need to get started, and even how to start debugging your code. I'll get into the basics of what the language is made up of; things like variables, types of data in JavaScript, how to handle text and numbers, and even how to let your code make its own decisions. In lesson two, I'll build on that, introducing more complex data types, and control structures, and how to make your code do things by creating your own functions.

LESSON 1




Tools, variables, data types, text, numbers, decisions

AQUENT
GYMNASIUM

JavaScript Foundations

LESSON 2




Advanced data types, control structures, functions

AQUENT
GYMNASIUM

JavaScript Foundations

In lesson three, you'll learn how to use JavaScript to interact with the HTML page itself by selecting elements on the page. Then, how to set and get various properties of elements, and how to set and get the style of any element on the page. Lesson four will show you how to create HTML elements and add them to, and remove them from, the page; and create, and use HTML templates to create, and dynamically populate an HTML page, or portion of a page.

LESSON 3



Selecting elements, changing properties and styles with code

AQUENT
GYMNASIUM

JavaScript Foundations

LESSON 4



Creating elements, adding and removing elements, templates

AQUENT
GYMNASIUM

JavaScript Foundations

Lesson five will cover sending data to, and receiving data from, a server, allowing your page to interact with the outside world. And finally, lesson six will take a look at what others are doing with JavaScript. I'll give a brief overview of some of the currently popular JavaScript application frameworks that are available, and then I'll dive a little bit deeper, and show you how to make a very simple application using the Backbone.js framework.

LESSON 5



Communicating with a server, sending and receiving data

AQUENT
GYMNASIUM

JavaScript Foundations

LESSON 6



Frameworks and libraries, sample app with Backbone.js

AQUENT
GYMNASIUM

JavaScript Foundations

don't expect this course to make you into an expert in any one of these subjects, but as a whole, it should give you a very solid foundation to be able to work with the language confidently, and build on your knowledge from there. So, before we dive in, in earnest, let's cover a little background history. JavaScript was created in 1995 by Brendan Eich at Netscape.

At that point, Java was already incorporated into browsers, allowing developers to deliver full scale applications, or applets, in web pages. JavaScript was intended as a lightweight alternative for adding smaller bits of interaction and functionality. Interestingly, it's rare to run across a Java applet these days, but full scale JavaScript applications are all over the place.

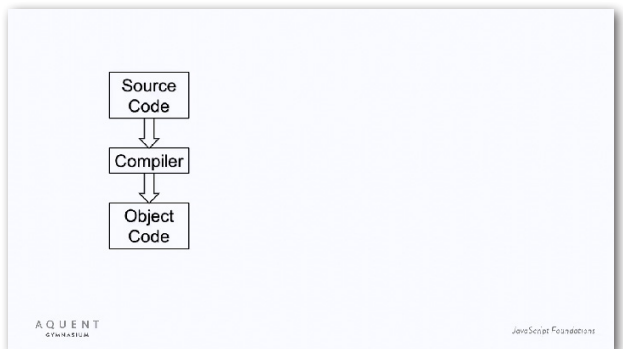
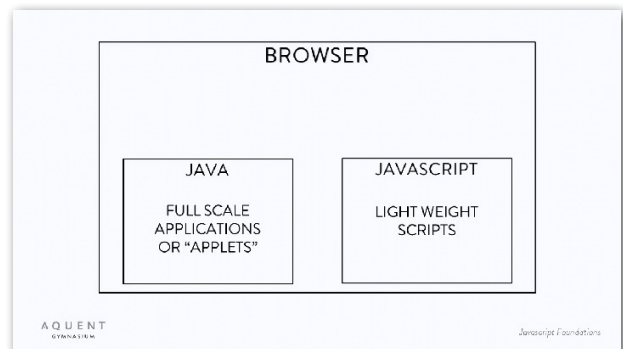
It's worth noting that beyond the two having similar names, there's no relationship at all between Java, and JavaScript. JavaScript is not a light version of Java. The two are completely different languages. Now, JavaScript is what is known as an interpreted language. What does that mean? Well, computer languages can be broken down into two categories; compiled and interpreted languages.

In a compiled language, like C++, you write the source code in a text editor, and then run that through a program called a compiler. This reads the code that you just wrote, and converts it to a special file containing what's called object code. This is your program in a format that the computer can read directly. Often, those bits are run through another program called a linker. This links your code with other pieces of code that it needs to function fully. The output of this is saved as the final, executable file. Now, you don't need to be too concerned with all that, for now, because that's not what JavaScript does.

As I've said, JavaScript is not a compiled language, but interpreted. This means that you write your code, and you save it in a text file, and you upload that file to a server somewhere, along with your HTML, CSS, and any other associated files. When a web page with JavaScript is loaded into a browser, the browser reads that JavaScript and interprets it into code that it can execute, right then and there.

In general, compiled programs usually have a lot more speed and power than interpreted programs. This is because, during the compiling and linking process, a lot of optimizations can be made to make the program run as fast and efficiently as possible. But in recent years, the JavaScript engines and browsers have come a long way in terms of speed and power. They can even do a certain amount of optimization during the interpretation process. If you used JavaScript in the distant past and think of it as painfully slow and clunky, you may be in for a surprise when you see what modern JavaScript can do in a modern browser.

Next up, we'll look at some of the tools you need to start writing JavaScript.



TOOLS FOR WRITING JAVASCRIPT

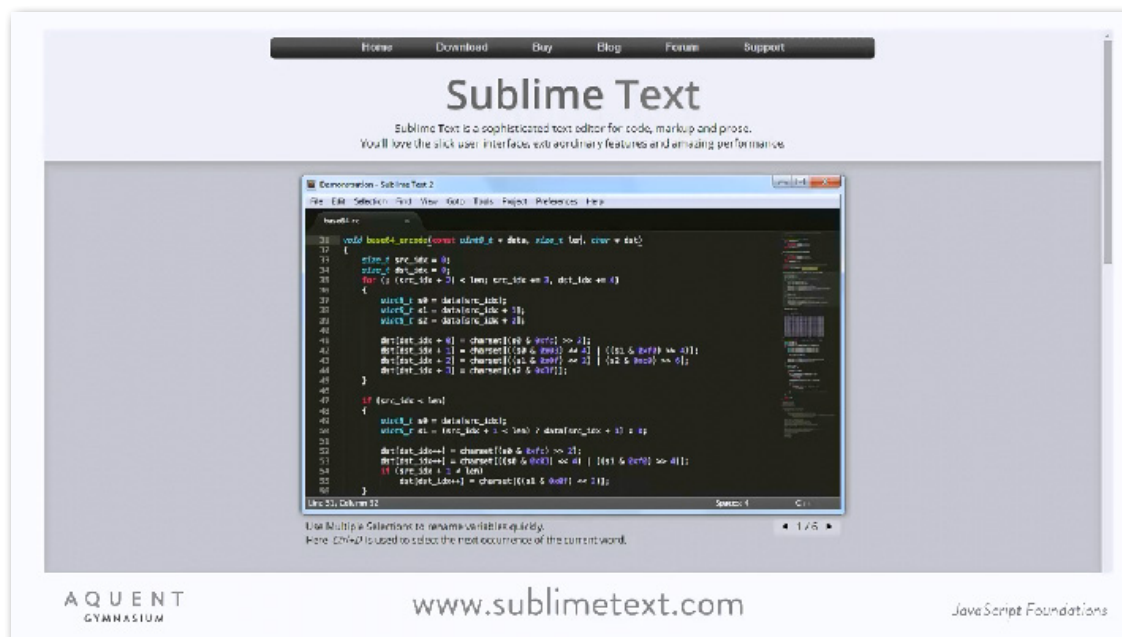
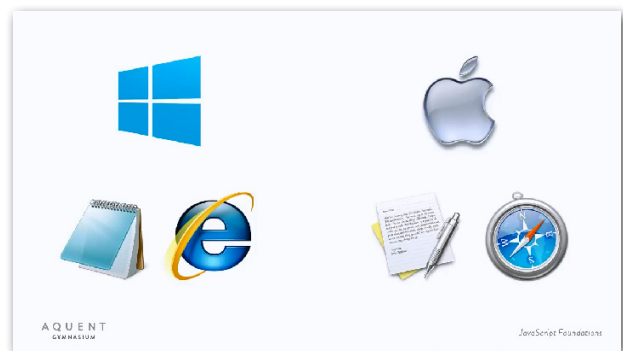
Like any other activity of creation, writing code requires certain tools. But, as I explained earlier, JavaScript is an interpreted language, not a compiled one. So, you're not going to need any special compilers, or linkers, or other complex tools.

All you really need is a text editor, and a browser, and I know for certain you have both of these on your computer. A Mac comes with TextEdit and Safari pre-installed, and Windows has Notepad and Internet Explorer. But, most likely, you're going to want to upgrade both of those, and expand on them.

First of all, you're probably going to want a text editor that's specifically designed for writing code. These give you all kinds of neat features, like colored syntax, highlighting, code completion, code templates, and snippets, and project management. There are many of these available on every platform, and I'm not going to go through all the possible choices.

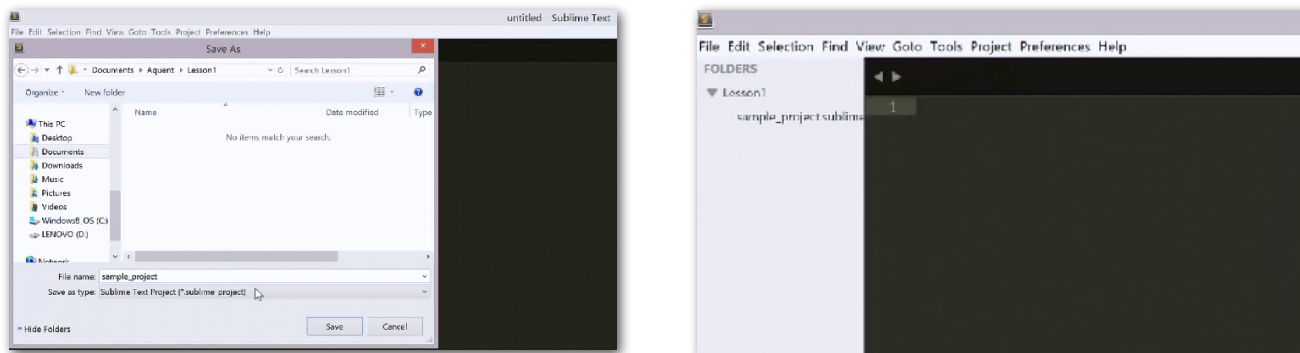
If you are already doing some kind of HTML and CSS work, you most likely already have some kind of editor that you're comfortable with. If this is designed to work with HTML and CSS, it will almost certainly handle JavaScript like a pro, so stick with that.

If you don't know where to start, I'm going to suggest, Sublime Text. That's what I'll be using personally for all the coding examples you see in this course. Technically, Sublime Text is not free, but you can download and install it, run it, and use all its features, indefinitely. Paying for it removes the occasional pop-up that it throws up, reminding you that it's not free.



Now, Sublime Text is a project-based editor, but creating a new project is a bit unintuitive, so, let's just go through it. If you have a project currently open, close it via the Project menu, Close Project. Then, select Project, Save Project As. Browse to a folder you want to store your project, and give the project file a name. Finally, select Project, Add Folder to Project, and add the folder where you stored your project.

Once you add a folder to the project, the sidebar will open, showing you all the files in your project. Sublime will start you out with an empty document. Save that, giving it the name, index.html. Make sure you save it in your project directory.



Now, because this is an HTML file, Sublime will enable some special HTML features. Type in the letters HTML, and hit the Tab key. Bingo. You have an HTML document, all ready for you to enter the title. If you go to the head section, and type link, and hit Tab, it will create a stylesheet link tag ready for you to enter the href. Similarly, if you enter script, and hit Tab, it will create a JavaScript tag that you can then, either, enter a source attribute, or just start writing some code.



Also, notice how the various elements and attributes are color coded. You see the same thing for CSS files, and JavaScript files. If you don't like the colors it's using, you can go to the Preference menu, select Color Schemes, and choose one of the many built-in color schemes. Or, if you want to go really crazy, you can create your own scheme. I'll be using the default color scheme and fonts for this course.

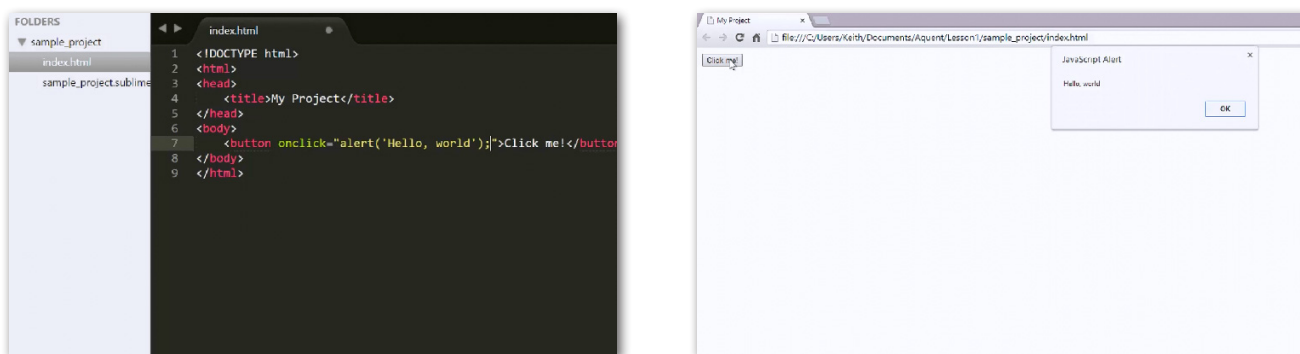
Now, the other tool you need is a browser. As I said, your operating system comes with a browser pre-installed, but you'll most likely want to install a couple more, so that you can test things out in different browsers; minimally, Google Chrome and Mozilla Firefox. But if you've done any HTML work, I'm telling you stuff you already know.

I'll be using Chrome for this course. It works well, and has a very nice set of developer tools that will help you inspect your code while it's running, and track down, and fix any bugs. We'll be covering those tools in the next section, so, I strongly suggest that for the purposes of this course, you should stick to Chrome as the browser that you use to test your examples with. That way, we'll both be on the same page.

But, I want to wrap up this section by discussing where you should actually put your code. There are three main options, and they're all very similar to where you would define CSS styles for a web page. For the rest of this section, I'll be writing and testing real code in an editor, and a browser. If JavaScript is very new to you, I urge you to stop the video, open up your editor of choice, or install Sublime Text, and follow along with what I'm doing. But if this is just review for you, feel free to just watch, for now.

The first option is called, inline JavaScript. You'll see this most often to deal with events like button clicks. Let's go over to that index.html document we just created. Go down to the body section, and add a button element. In Sublime, I can just type button, and hit the Tab key. Then, I'll give it some text to display.

Now, I want something to happen when I click on this button, so I'll add an onclick attribute, and assign a text string to it. That text will be actual JavaScript code. Let's call up an alert box with the text, "Hello, world", in it. So, I type onclick equals quote alert "Hello, world."



Now, I can launch this file in Chrome, and you can see that it created that button. When I click on the button, the JavaScript I wrote is interpreted and executed. The alert box is displayed with the message I gave it.

So, this works, but you almost never, ever want to do something like this. It's very much the same thing as using inline styles. We could add a style attribute to this button, say, color colon red, and as expected, you will see that this changed the color of the text on the button to red. But, you would probably never do that for styles, either.

Say, you wanted all your buttons to have a certain style, which included red text. But, later, someone decided that they should have a slightly different shade of red. If you used inline styles, you'd have to hunt down every instance of every button that was styled this way, and change them, one by one.

It's the same for code. Sites can get large, with many HTML files, and all kinds of complicated code controlling it all. You don't want to be scanning through HTML file after HTML file, trying to find inline scripts. It's much better to centralize all your code, the same way you would centralize all your styles.

So, the next option is the script tag. This is equivalent to a style tag in an HTML page. With a style tag, you write all your CSS inside that tag, and it gets applied to anything on that page. With a script tag, you write JavaScript code right inside the script tag, and it gets executed when the page loads.

Back in the code, I'm going to clear out all the button attributes I created before, and give this button an ID of btn.

Then, I'm going to add a script tag right below this. Now, you might be used to seeing script tags up the head section of an HTML document. And yes, that's normally where you'd see them. I'm going to put this one after the button, because the document is going to load it in the order it's written, and I want the button to exist before I start writing code that deals with it. In lesson three, you'll learn a lot more about this, so I wouldn't worry about it too much, right now.

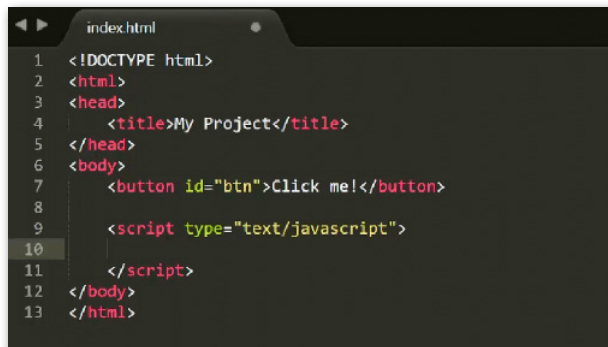
I'm going to write some code in here that's going to find this button via the ID we just gave it, and assign a JavaScript function that will run when the button gets clicked. This will contain the same line we wrote before, alert("Hello, world!"), and we can test that. And, yes, the function gets called, and the alert box appears.

If you don't understand a single thing about the code I wrote, do not worry about it. The actual code, here, is not the important part. After you get through a couple more lessons, you'll understand it completely. For now, the only part I really want you to grasp is that we move the code out of the button tag, and into a script tag.

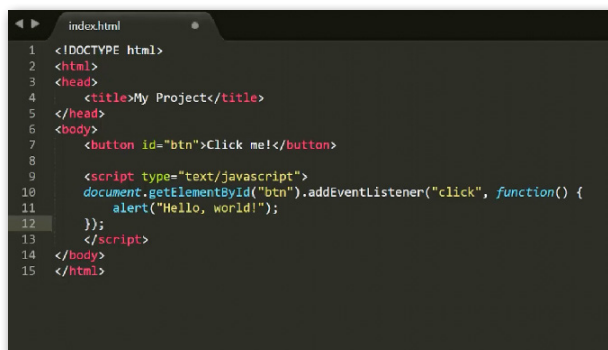
If we had other code that did other stuff, they would also go into the script tag. All the code for this page would go into that one tag. Then, if you needed to find the code for this page, you could go straight to the script tag, and it would all be right there.

So, like inline styles versus style tags in a page, script tags are a huge improvement over inline JavaScript, but it's still not perfect. To see why, let's create a style tag, and set a style for the button element that gives it a color of red. Great. So, now, every button on this page will have red text.

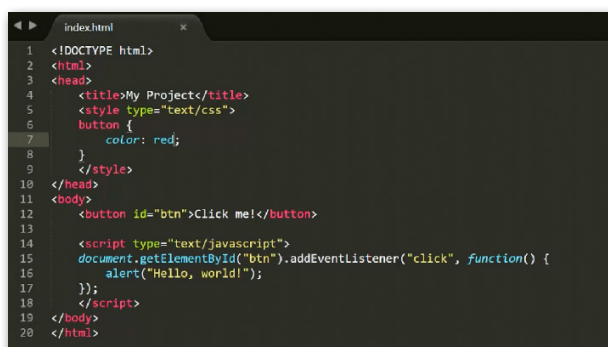
But, say we navigate to a new page, within the same site. Buttons are back to normal black text. We'd have to copy this style tag into every other HTML page on the site. Then, if we want to change the color, we have to go through every single page and change that style.



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>My Project</title>
5 </head>
6 <body>
7   <button id="btn">Click me!</button>
8
9   <script type="text/javascript">
10
11 </script>
12 </body>
13 </html>
```



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>My Project</title>
5 </head>
6 <body>
7   <button id="btn">Click me!</button>
8
9   <script type="text/javascript">
10     document.getElementById("btn").addEventListener("click", function() {
11       alert("Hello, world!");
12     });
13 </script>
14 </body>
15 </html>
```



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>My Project</title>
5   <style type="text/css">
6     button {
7       color: red;
8     }
9   </style>
10 </head>
11 <body>
12   <button id="btn">Click me!</button>
13
14   <script type="text/javascript">
15     document.getElementById("btn").addEventListener("click", function() {
16       alert("Hello, world!");
17     });
18 </script>
19 </body>
20 </html>
```

It's the same with JavaScript. You'll very often be creating bits of code functionality that you want to have exist on multiple pages. Copy and paste is not a viable solution for reuse, so we centralize our styles in code across the whole site. We create site-wide CSS files that are referenced by many, or all, of the pages in a site. Likewise, we create external JavaScript files that are accessed from any page that needs the functionality contained within that file.

So, back in our HTML file, I'll continue to use this script tag, but I'll delete the actual script out of it. Instead, I'll add a source attribute to it, and point that to an external JavaScript file. I'll call it `main.js`. JavaScript files generally have a `.js` extension by convention, but technically, they could be named anything.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>My Project</title>
5 </head>
6 <body>
7   <button id="btn">Click me!</button>
8
9   <script type="text/javascript" src="main.js"></script>
10 </body>
11 </html>
```

Now, even though this script tag is empty, it's very important that you don't shortcut it like this. While that is a valid way of closing an empty tag in HTML, it won't work for linking to external JavaScript files. You need to add the full ending tag.

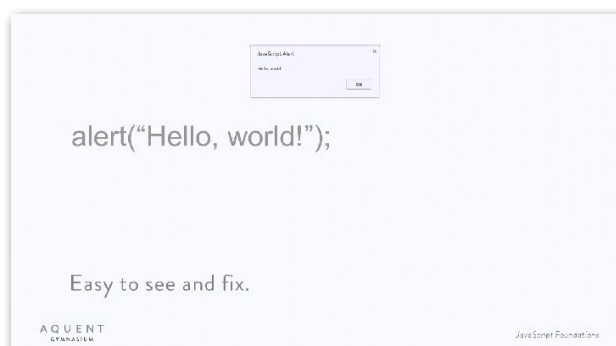
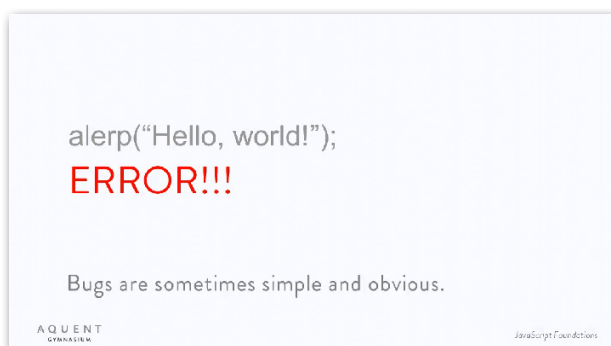
Now, I'll create this `main.js` file by creating a new file in Sublime Text, and saving it with the name, `main.js` in the project folder. Then, I'll just write the same basic code that I had in the script tag earlier, and save it again. Test that in the browser, and we're still in business.

Now, hopefully, this all makes sense to you, because every bit of code I'm going to write for the rest of this course will be in external JavaScript files like this. I think you'll get very used to it, very soon.

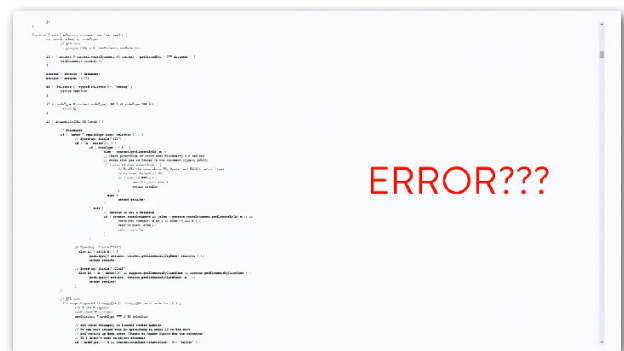
DEBUGGING JAVASCRIPT

Next up, I want to cover some very basic debugging techniques. It may seem odd to cover debugging before you even learn the first thing about the code itself, but bear with me. Debugging, as its name implies, has to do with finding the bugs, or problems, in some code, and fixing them. Here, you can see one of the first documented computer bugs. It was literally a bug; a moth that had gotten stuck in a mechanical relay.

Now, initially, the programs you create will be very simple, and the errors that you make will likely be misplaced symbols, or misspelled terms. They'll jump right out at you when you take another look at your code, and they'll be easy to correct. But, as your programs get more complex, the areas can get more subtle, and have more to do with the sequence of actions, or some missing step. You're not likely to spot this just by scanning through the code itself.

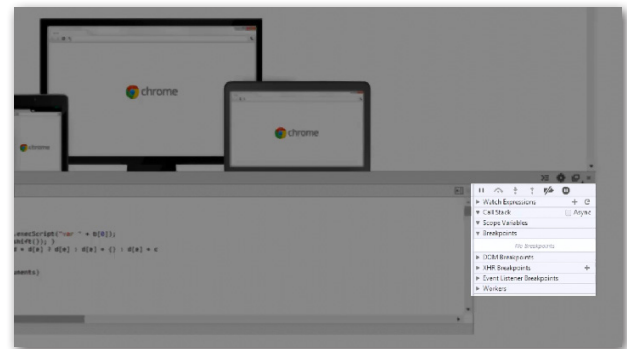


The solution is to be able to inspect what the code is doing, as it is running. Fortunately, most browsers now have fantastic developer tools built right into them that help you do this. So, when I said a browser is one of the tools you need to develop JavaScript, I wasn't only talking about the ability to be able to look at the final product. The tool set included in the browser actually includes a rich set of debugging tools, very much like what you'll find in high-end developer packages like, Visual Studio, Xcode, or Eclipse.



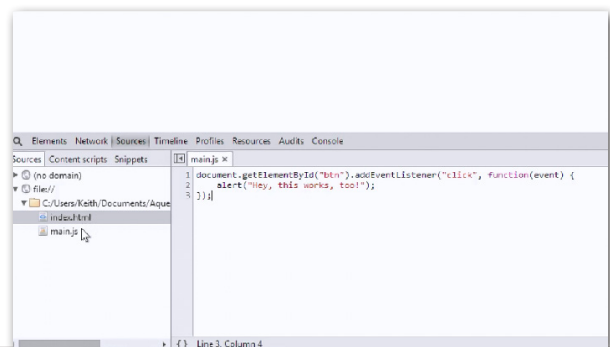
With the developer tools, you can write statements in your code that will get logged to a special panel in the browser called the console. Then, by watching the console while your program is running, you'll see the message you wrote, along with any values you might have sent to it. Perhaps you were expecting some value to equal 10 at a certain point in the program, but when you log in you see that it's actually nine. This can often be an “aha moment,” when you realize where things went wrong.

There are even more advanced debugging tools that allow you to pause your program when it hits a certain point. You can then go in and look behind the scenes to see exactly what's going on at that point, and even step through the code, line by line, and see exactly where things go wrong. We'll cover that kind of advanced debugging in a later lesson. For now, I just want to introduce you to the developer tools, and in particular, to the console. We'll be using the console extensively as we go through the parts of JavaScript.



So, go ahead, and open up your browser. Actually, since you're probably watching this online, you most likely already have a browser open. In Google Chrome, to open the developer tools, you can go to the Tools menu, and choose Developer Tools, or press Control Shift I in Windows, or Command Alt I on a Mac. You can also press the F12 key on a PC. These shortcuts generally hold true for other browsers, such as Internet Explorer or Firefox, as well, but you may need to verify what the shortcut is, if you use some other browser in the future. This will open a panel at the bottom of the browser that looks like this. If you've been doing HTML and CSS work, you may already be familiar with this.

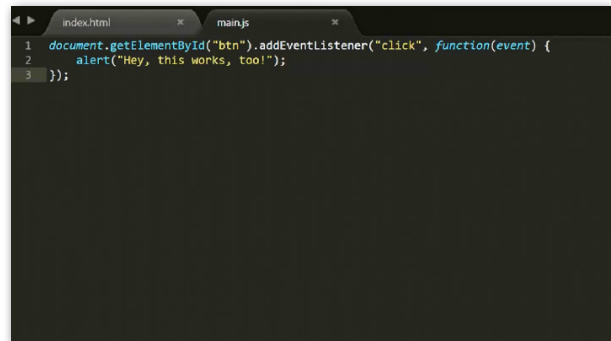
The elements tab of this panel shows you the HTML structure of the page currently loaded in the browser. You can select any element, and see all of its styles and metrics in this panel, here on the right. While doing JavaScript development, you'll mostly be focused



on the sources tab in the console. The sources tab shows you all the files that make up the current page.

Here, you can see index.html and main.js. If you click on the main.js file, you can see the actual code that is loaded into the browser. In lesson two, I'll show you how to use this tab to inspect, and control your code at run time, in order to see what's happening and fix any bugs. For now, we'll be looking more at the console. The console will show you any errors or warnings that occur while your JavaScript program is running. But, you can also send your own messages to the console, right from within your program.

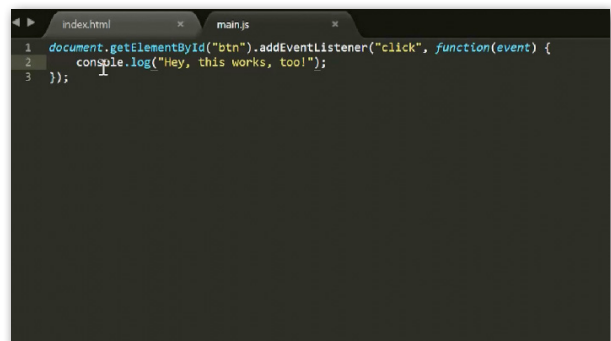
Now, looking back at our main.js file, you recall that I used the alert function to display an alert box. This is often the first thing new JavaScript developers do, when they need to debug something. You type in alert, followed by open and close parentheses. Inside the parentheses, you put any message you want to display in the alert box. If you are uncertain that a button was responding to clicks, for example, you could put an alert here, just like I did in this main.js, and this is going to tell you for sure, whether or not your button is working. If the alert box didn't show up, you know something was wrong right there.



Now, while alert works okay for this purpose, it's not really ideal. It's obtrusive, and it freezes the web page until you cancel it. Here, you can see that while the alert box is open, not only is the web page itself disabled, but the entire browser is frozen. I can't click any of these buttons in the UI, or select the URL text, or anything.

Logging messages to the console works behind the scenes. The program, and entire page keep running as usual. Most visitors to the site will never even know that a message has been logged, but it will be right there when you open the console.

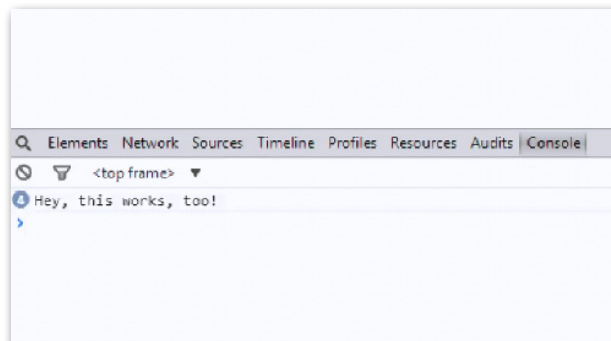
To see this in action, go back into the main.js file, and replace the word "alert" with "console.log." Here, console is a JavaScript object that represents the console tab in the developer tools. You'll learn more about objects in the next lesson.



Log is a function on that object. It works almost exactly like alert. You pass it a message by putting it in between the parentheses.

So, save that main.js file, and open up or reload that index.html file in the browser. Make sure your console is open, and click that button. You should now see the message appear in the console.

If you click the button a few more times, you might



expect to see the message appear again, but instead what happens is that a small number appears to the left of the message. This tells you that, that exact message has been logged that many times. Sometimes in a program the same message might wind up getting logged dozens, or hundreds, or even thousands of times. Rather than make you scroll through page, after page of the exact same message, you could just see how many times that particular message was logged. So, throughout the rest of the course, you'll be using the console extensively to see how the different parts of JavaScript function, and see exactly what's happening in the applications you'll be writing.

Finally, I want to show you one more thing about the console. First, let's go back to the code, and just change the spelling of console; spell it with a "k." Now, this is an error, a typo. There is no object named, `konsole` with a "k."

Well, let's see what happens when we run this. Save the file, go back into the browser, and reload the page. Now, with the console open, click that button again. Now, instead of your message, you'll see another message displayed in red. Mine says "uncaught reference error, `konsole` is not defined."

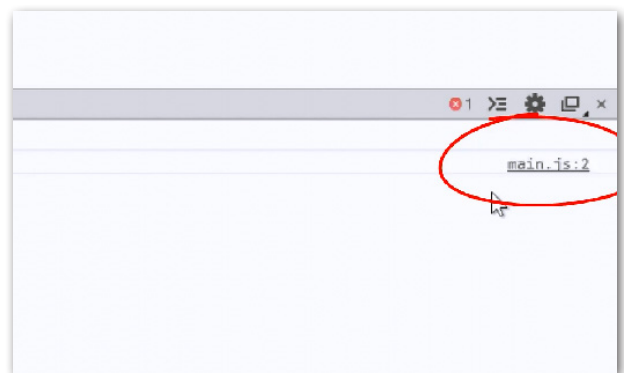
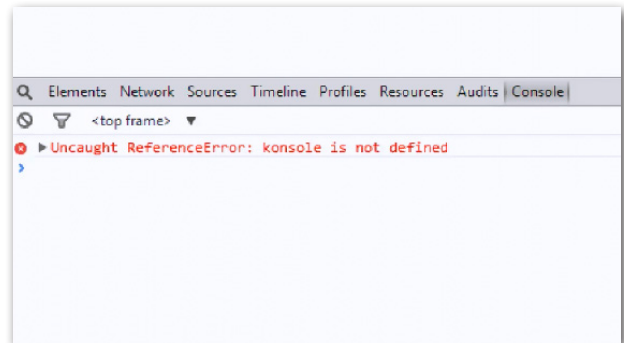
Now, you may, or may not know exactly what this means, but the part about `konsole` not being defined should give you at least a clue. But, more than that, look over to the right here, and you'll see this line `main.js` colon 2. This means that the error is in the `main.js` file, and that it's on line two. Fantastic. It's like having an invisible senior programmer standing over your shoulder, pointing you to the exact line, in the exact file where you made an error.

So, you go back to `main.js`, and you look at line two, and you see that you misspelled `console`, aha. Set it back to `console` with a "c," and you've done your first bit of debugging. Save it, reload the file, and ensure that it now works again.

Now, there's a lot more to debugging than that. But to be honest, many, many developers get by their entire careers using only what I've just covered. I'm going to push you right past all of them in debugging ability by the end of lesson two. But in the meantime, what you know now, will get you very far.

VARIABLES

It's hard to know exactly where to get started in describing an entire language, but I think one of the most fundamental building blocks is the concept of a variable. A variable is like a container for data. It has a name and a value. You can create a variable at any time by typing the keyword "var," followed by the name you want to give the variable.



Let's go into the project we've been using, and try this out. Again, if this is somewhat new to you, actually open up your editor, and type in what I'm typing. I'm going to clean things up here, remove this button, and move the script tag up to the head, where you're probably more used to seeing it. In the main.js file, I'm just going to delete everything here. At this point, you can say you completely understand everything in this file. My hope is that you will continue in that state for the rest of the course.

First, let's create a variable. I'll just type, `var message`. Now, I have an empty variable named `message`, ready to store some data. Note, this semicolon on the end of that line. In JavaScript, every statement is supposed to end with a semicolon.

Now, let's put some data in that variable. Next line, `message` equals `hello, world`. You assign data to a variable with the equal sign. This is known as the assignment operator. You put the variable on the left, and the data you want to store on the right. The data I'm storing is the text, `hello, world`.

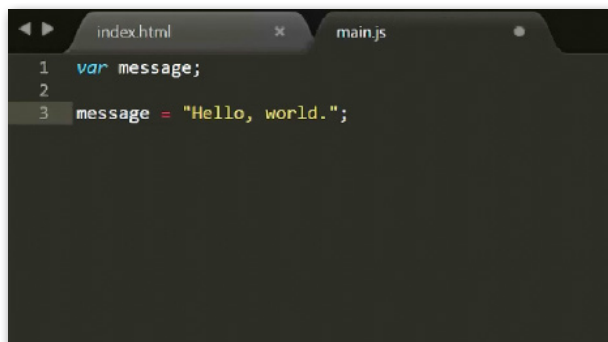
Anytime you use text in JavaScript, you need to surround it with either single, or double quotes. I could have used single quotes here, like this; absolutely no difference. Text surrounded by quotes is called, a string in JavaScript. The string is one type of data. It's obviously used for storing text, and you'll learn a lot more about strings later in this lesson.

So, what can we do with this `message` variable? How about logging into the console? I'll type `console.log(message)`. So, there, I passed a variable instead of a raw string. If I save that, load the page in the browser, and check the console, sure enough, it says `hello, world`.

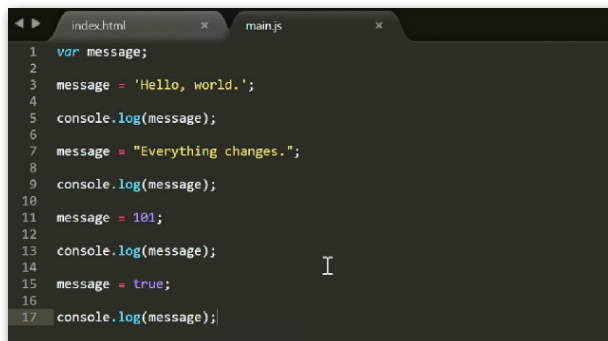
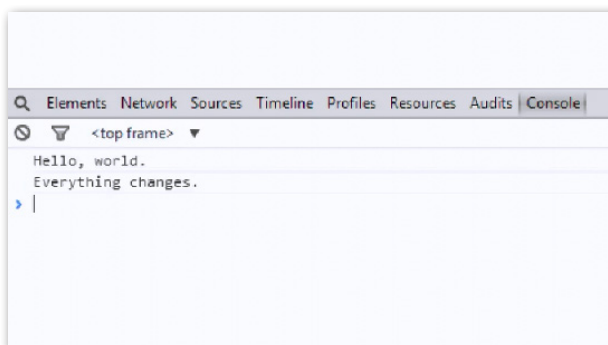
Now, the name "variable" implies that the data in the variable can change, or vary. And, sure enough, it can. I'll go back in, and after that log statement, I'll change the message to, `everything changes`, and I'll log the message, again. Save, reload the page, and check with the console. So, it shows both messages. Great.

Now, let's change it again. We'll say, `message` equals `101`. Here, I've assigned a number to the `message` variable. Numbers are another data type in JavaScript. Generally, you just type the digits of the number, no quotes. I can log that again with the same log statement, and check the console, and it still works as expected.

But, this brings up a very important point. In JavaScript, any variable can hold any data type. The mes-



```
1 var message;  
2  
3 message = "Hello, world.";
```



```
1 var message;  
2  
3 message = 'Hello, world.';  
4 console.log(message);  
5  
6 message = "Everything changes.";  
7 console.log(message);  
8  
9 message = 101;  
10 console.log(message);  
11  
12 message = true;  
13 console.log(message);
```

sage variable doesn't care if it's assigned a string, a number, or anything else. The last data type I'll tell you about right now is Boolean. This is a value that can be either true or false. So, I can say, message equals true.

Note that I didn't put any quotes around that. It's not a string. It's not text. True and false are special words in JavaScript, and can just be typed, as is. Again, I'll be covering a lot more about these three data types I just mentioned, later in this lesson. But, for now, it's enough that you know they exist. So, we can log that, and sure enough, it logs true.

Okay, so, now you know how to create variables, and how to store information in them, and even a little bit about how to use them. The act of creating a variable is also known as declaring a variable. A shortcut that's very often used is to declare a variable, and assign it at the same time. I'll clear all this out and re-declare a message, and assign it some data, at the same time. So, if you know what data a variable is going to hold when you declare it, you can save yourself a line of code.

Now, say, you were making some kind of registration page for a site. You'd probably want several variables to hold the various bits of data about the person who's registering, so, you might make a few variables like this. That's fine, but there's another shortcut you can do, here. You can just use the var keyword once, and then list all the variables after it, separated by commas.

Usually, to make this more readable, people put each variable on a new line, indented to line up, like so. Note, that the semicolon still goes on the end of all that. You can also assign data here, as well. You can assign data to all the variables, or none of them, or any combination. Here, I've assigned strings to the first and last names, and I've left age empty. That's all good. You can then assign age, later.



```
1 var firstName = "John",
2   lastName = "Smith",
3   age;
4
5 age = 39;
6
7 console.log(firstName, lastName, age);
```

Now that you have multiple variables, you should know that you can log multiple variables. Just list them with commas inside the console.log function. And, here, in the console, you can see that those three values are logged all in one line.

The last general thing I want to tell you about variables is, how to name them. Rule number one: a variable must begin with an uppercase letter, a lowercase letter, an underscore, or the dollar sign. The rest of the characters in the name can be any of those same characters, or numbers. So, all of these are valid variable names, and these are not. Technically, there are other special characters that you can use, but people usually only use them when they're trying to be cute, or impress someone.

Rule number two: variables can not have the same name as reserved keywords in JavaScript. JavaScript has a list of words that it uses for the language itself, and has reserved a few more for possible future use. You can't use those as variable names. You can find a list of reserve keywords, [here](#).

Now, those are the only hard and fast rules. If you don't follow them, your code may crash or not run at all. The next rule is really just a convention or best practice.


```
var firstName;  
var LastName;  
var _item7;  
var $age;
```

Rule #1: Begin with a-z, A-Z, _ or \$.
Continue with a-z, A-Z, _, \$ or any number.

AQUENT
GYMNASIUM

JavaScript Foundations

```
var function;  
var return;  
var var;  
var super;
```

Rule #2: Don't use reserved words.

AQUENT
GYMNASIUM

JavaScript Foundations

Rule number three: generally, variables are named with what we call camel case. This means that they start with a lowercase letter. Single word variables are composed of all lowercase letters. And, variables that contain multiple words, like first name or day of month, have the first letter of each subsequent word, capitalized. Now, some developers use underscores to separate words like this. This is valid, but it's far less common.

Finally, the last rule is really just a suggestion, and common sense, but it needs to be stated. Rule number four: use variable names that actually mean something, and don't abbreviate when it will make the meaning unclear. Someone else may eventually need to read, and understand your code, someday. That someone may be you, months after you wrote it.

```
var name;  
var firstName;  
var dayOfMonth;  
var bestThingEver;
```

"Rule" #3: Use "camel case".

AQUENT
GYMNASIUM

JavaScript Foundations

```
var first_name;  
var day_of_month;  
var best_thing_ever;
```

"Rule" #3: Use "camel case".
Underscores are valid, but less common.

AQUENT
GYMNASIUM

JavaScript Foundations

A single letter variable name, like `f` or even `fn`, may leave you scratching your head wondering what it's for, whereas, `firstName`, leaves no question. Vague, variable names like `thing` or `data` or even `num` for numbers and `str` for strings seem better, but they're really just as meaningless as single letter variables.

Now, before I go any further, I'll be showing you a lot of small examples in the rest of this lesson. It would be a bit annoying to constantly switch back to the code window, type in some code, save it, switch back to the browser, reload, and check the console, so, I'm going to start using a shortcut. Let's open the console here, and take another look at it.

Down here, you see a greater than arrow with a blinking cursor. That is a command line prompt, where you can enter JavaScript statements, and they'll be executed immediately, and any results will be displayed right there.

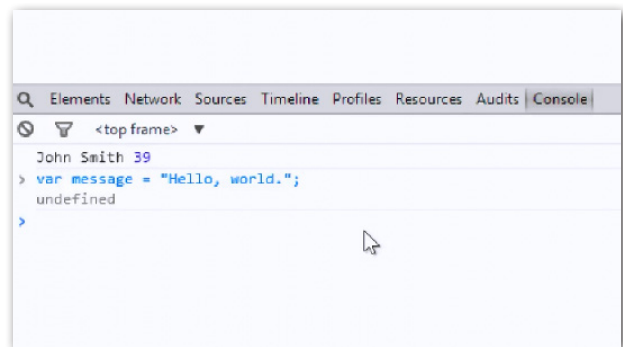
For example, I can type `var message equals hello, world`, and press Enter. Now, that prints out `undefined`, which might make you think that you did something wrong, but you didn't. Whenever you declare a variable,

the console is going to say undefined, so don't worry about it. There are other statements that will do this, as well.

To verify that you actually did declare, and assign a value to the message variable, now, type message, and you see that it shows the value you assigned. Typing any variable name will display the value of that variable immediately. Now, try typing alert(message), and you see the alert box appear with the value of the message. So, you could actually run just about any JavaScript statement, here. And, it's much quicker than using an editor, and a web page for testing simple examples like this.

Now, there are a couple more data types that you should know about, undefined and null. If you see an error in the console, and it mentions something not being defined or being undefined, it generally means that you tried to do something with a variable that was either, never declared, or never assigned a value.

Now, this can be a timing issue. You're trying to use a variable that hasn't been defined or assigned, yet. Quite often, though, it's just a typo. For instance, I've defined a variable called message, but if I make a mistake and spell it massage, the console is going to give you an error saying that massage is not defined. However, if you declare a variable with that name but don't assign it a value, and then check that, there is no error. But, it will tell you that it's undefined.



Now, the null value is similar to undefined. Null also means no value. But in JavaScript, it's generally purposefully assigned. Sometimes, you declare a variable and you don't have a value for it yet, but you don't want to leave it undefined, so you can say message equals null, and then check that.

Okay, now you know what variables are, and we're ready to take a bit of a deeper dive into some of the different variable types, and the kind of operations that you can do with them. We'll start with numbers and Booleans.

NUMBERS, BOOLEANS, AND CONDITIONALS

Numbers are a pretty basic data type, and in JavaScript, they're drop dead simple. In many languages, there are several different types of numbers; tags for integers and decimals, large numbers, small numbers, et cet-

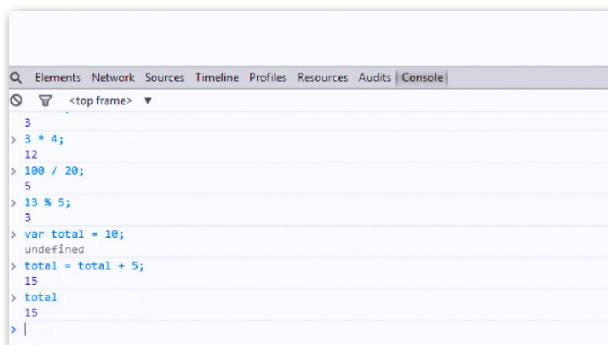
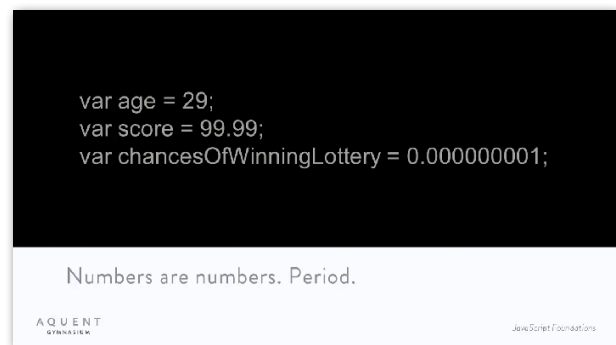
era. In JavaScript, a number is a number. You specify it just by typing the digits.

Now, let's try some live coding. Open up your console, and as I go through these examples, try them out for yourself. And, feel free to pause the video, and try out some other examples on your own.

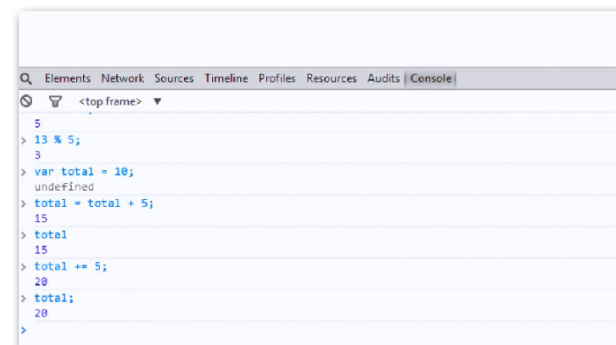
So, all normal mathematical functions, you'd expect to work with numbers, and generally do; addition, subtraction, multiplication, division. There's also the percent sign, which is the modulus operator, also called the division remainder. If you divide 13 by five, you'll get two, with a remainder of three. Two times five is 10, plus three is 13. The modulus operator gives you that remainder.

Now, sometimes you want to add a value to a variable, and store the result back in that variable. You could say `var total equals 10`, and then, `total equals total plus five`. So, you're adding five to `total`, and then storing the result, 15, back in `total`.

But, there's a shortcut. The add and assign operator, just say, `total plus equals five`. This adds five to whatever is in `total`, and stores it back there. So, now a `total` contains 20. These type of operators exist for subtraction, multiplication, division, and modulus, as well.



A



common action is to keep track of how many times something has happened. You can set a variable like `count` to zero, and then, each time something happens add one to it by saying `count plus equals one`. But this is so common that there's a shortcut; plus plus, the increment operator. So you can say, plus plus `count`, and that adds one to whatever is in `count`.

There's an alternate version, where you put the plus plus after the variable. This does the same thing, but the timing is a bit different. When you use the plus plus before the variable, it's called a prefix operator. This will add one to the variable, and then, that variable will be used in whatever statement it's contained within. If the plus plus comes after the variable, it's called a postfix operator. The original value of the variable will be used in the statement, and after that, one is added to the variable.

So, right now, the value of count is two. If I type, count plus plus, the console logs the value of two and then adds one to it. If I type, count again by itself, you can see that it actually did get incremented. Try this a few times on your own, until you fully understand it.

Now, math and JavaScript generally follow all the same rules for orders of operations. So, for one plus one times two, you will do the multiplication first, one times two equals two, and then the addition, one plus two equals three. And, you can use parentheses to change the order of operations. Here, I'm saying one plus one equals two first, then two times two equals four. And, that's about all there is for numbers, themselves.

But, if you're going to be doing a lot with numbers, you'll find the math library in JavaScript very handy. This is a collection of common, and very useful math functions. You access it by typing math, dot, plus the name of the function.

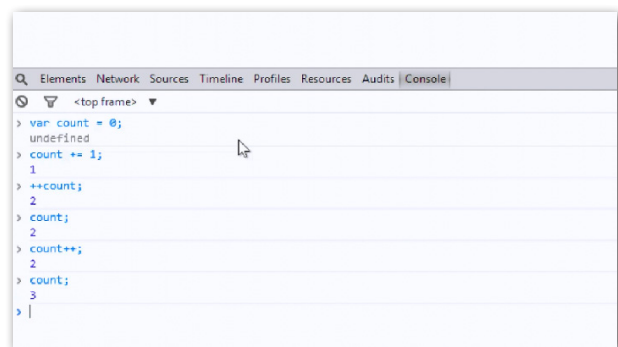
For example, the math dot max function takes two or more numbers, and returns the value of the larger one; so math max five, 10 is 10. Math SQRT gives you the square root of a number. And, there's math round for rounding numbers to the nearest whole number. And math random gives you a random number between zero and one. If you want a number in a higher range, like zero to 100, just multiply the random number by 100.

There are many more math functions, including a full complement of trigonometric and logarithmic functions, in case you ever need to do any more serious math. You can check out the documentation for the math object here, if you want to know what else is there. So, that's a quick look at numbers in JavaScript. Next up, we'll cover Booleans and conditional operations.

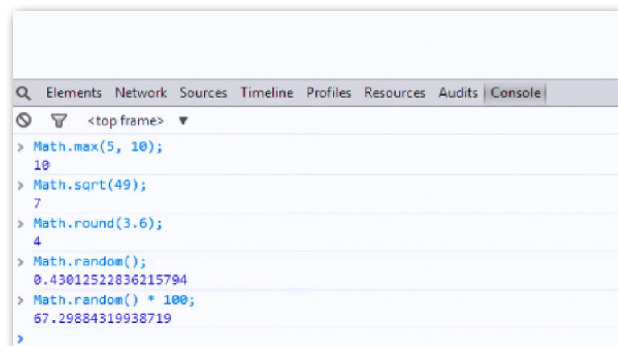
BOOLEANS AND CONDITIONALS

As mentioned earlier, Booleans are simply, true and false values. As such, there's not much more to really say about them, than that. What's more important is how you can get them, and what you can do with them.

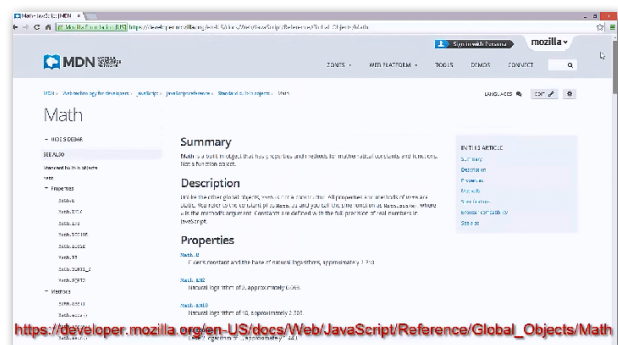
First, let me introduce you to a group of operators that let you compare two values. These will all return a true or false value. First, is the equals operator. Five equals five is true. Five equals six is false. Note, that this is a double equals sign, so it is different from the single equals assignment operator.



```
<top frame>
> var count = 0;
undefined
> count += 1;
1
> ++count;
2
> count;
2
> count++;
2
> count;
3
> |
```



```
<top frame>
> Math.max(5, 10);
10
> Math.sqrt(49);
7
> Math.round(3.6);
4
> Math.random();
0.43012522836215794
> Math.random() * 100;
67.29884319938719
> |
```



And, then there's the not equals operator. Five not equals six is true. Five not equals five is false.

Then, there are greater than, and less than operators. Six greater than five is true; five less than four is false. And, you can combine those with equals, so that five greater than, or equal to five, is true.

You can also use any of those to compare strings. So, a rose is a rose. If two strings contain the same text, they will be equal. If they have different text, they won't be equal. So, a cat is not a dog.

But, what about the other comparison operators with strings? Is a cat greater than a dog? Well, there you go, dog lovers. You were right, all along. Actually, what's going on here is an alphabetical comparison. D comes after c in the alphabet, so the string dog is greater than the string cat.

But, be careful. The way characters are coded in JavaScript, all capital letters actually come before all the lowercase letters. So a lowercase cat is actually greater than an uppercase dog.

And, of course, true equals true, false equals false, but true does not equal false.

So, now you can compare two values and see whether or not they're equal, and get a Boolean value from that. But, what can you do with it?

Well, here's where we actually start to get into some real programming. Enter the if statement.

Computers are great at storing data. We've seen that with variables. And, they're great at processing data, as we've seen with all the various calculations you can do with numbers. Another thing they're really good at doing, is analyzing and acting on that data.

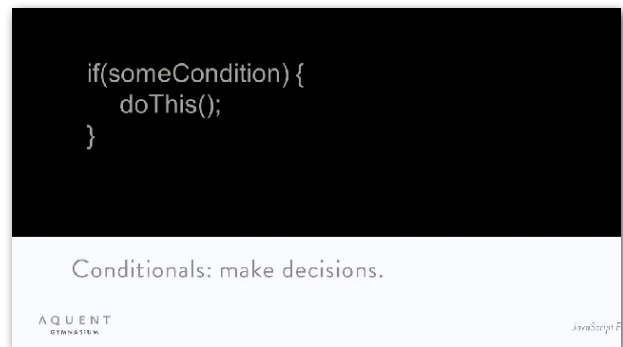
If you enter your username and password correctly, the system lets you onto the site. Otherwise it locks you out. It analyzed the data you entered, and either accepted or rejected it, and acted accordingly. A lot of that is accomplished with the if statement.

The way an if statement works is, like this. You say, if (followed by a pair of parentheses), with some value or variable, or some expression inside them. This is followed by open and closed brackets. Inside those brackets is some more code. So, if the thing in the parentheses evaluates to true, then the code in the brackets will run. If it evaluates to false, that inner code will be skipped.

To demonstrate this, I'll jump back into the code editor, because this is going to be more than a simple one-liner.

Now, this is the same HTML file we had before. It simply has a script tag that loads a file called, main.js. Right now, main.js has nothing in it. Now, a stupidly simple example is simply saying, if true. Now, of course, true will always evaluate to true, so that console log statement will always run.

A more realistic example would be to have a username passed in from some HTML text field. I'll simulate that



with a variable, for now. Now, of course, a real login code for a real site would be a bit more involved than that. But, somewhere in there, I guarantee you that there's an if statement, not unlike this.

As it's written, you'll get into that if statement, and a console log statement will run. If you change the value of username that code will not execute. You can use the if statement with numbers in comparisons, as well. So, if I set age to 35 and minAge to 18 and then compare the two with greater than or equals, because age is greater than minAge, you get in.

Now, you might be thinking, what if age is less than minAge? You could make another if statement saying, exactly that. But, there's an easier way, using an else statement.

The else statement looks like this. So, we have a block of code after the if, then an else, and then another block of code. One, or the other of those two blocks will always run. If the if condition is true, the first block will run, and the second won't. If the if results in false, then only the second block will run.

So, in the code you can change the second, if statement into an else statement. So, if age is greater than or equal to minAge you get in; else, you don't.

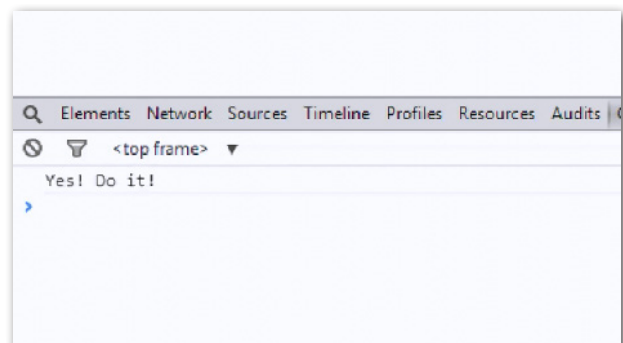
Using this, you can make a very sophisticated decision-maker. First, we'll generate a random number, and compare it to 0.5. The random number will have a 50 percent chance of being above 0.5, and a 50 percent chance of being below. So, roughly half the time the first block of code will run, and half the time the second one will. Ask it a question, refresh the page, and get some good, solid advice.

Should I sleep in today? Okay. Should I handle my email inbox today? Hmm. Not sure I like this program. There needs to be more options.

```
1 var userName = "user1235";
2
3
4 if(userName == "user1234") {
5   console.log("Welcome to the site");
6 }
```

```
1 var age = 35,
2   minAge = 18;
3
4
5 if(age >= minAge) {
6   console.log("Welcome to the site");
7 }
8 else {
9   console.log("Sorry, too young.");
10 }
```

```
1 var rand = Math.random();
2
3 if(rand > 0.5) {
4   console.log("Yes! Do it!");
5 }
6 else {
7   console.log("NO! Don't!");
8 }
```



Well, there is an, else if statement line you can throw in there. This looks like this. And, it works just about how you would expect it to.

If this, do this. Else, if that, do that. Else, do the other.

So, let's add a maybe clause to the decision-maker. Here, I'm breaking down that random number into three, roughly, equal chunks. If it's greater than 0.66, you get the first option. Else, if it's greater than 0.33, you get the second. Else, it must be less than 0.33, so you get the third choice, maybe.

You can actually throw as many, else if blocks in there, as you want. You can then finish it up with a single, else block, or just leave that off. With this structure alone, you can create all kinds of complex logic.

Now, the last thing I want to mention here is a concept called, type coercion. This means, that if JavaScript is expecting a certain type of data and gets a different type, it will try to coerce, or convert, what it got, into what it needs. This can be really useful in, if statements.

A common use for this, is dealing with undefined or null variables. As you saw in the last video, trying to do things with undefined variables can result in errors. You can get similar errors trying to use null values. So, if there's any chance that a variable might be undefined or null, it's good to check it before using it.

Say, you have a shopping cart application. And, there's a cost variable, which is 10, 10 dollars, or whatever. And, a quantity variable, which is not yet set. You'll be multiplying cost by quantity to get a total. But, if quantity is never set, you'll be in trouble. So, before you do the multiplication, you can check to make sure that quantity is not undefined.

If that's the case, you can safely calculate the total. Else, you can display an error. But, there's a shortcut you can use here. Just say, if quantity. The if statement is expecting a Boolean value, true or false. But, instead it gets either a number or undefined.

If it gets a number other than zero, it will coerce this into true, and the first part of the if statement will execute. If it gets a zero, or undefined, this will be coerced into false, and the error will be displayed.

Now, when we run this, we get an error, because quantity is undefined. If we set quantity to a number like two, the result is 20.

So, what gets coerced into what? Well, there's a list of rules. But, once you get them, they're pretty logical, and easy to remember. Undefined and null get coerced into false. For numbers, zero is coerced into

```
if(someCondition) {  
  doThis();  
}  
else if(anotherCondition) {  
  doThat();  
}  
else {  
  doTheOther();  
}
```

Else if: more complex decisions.

AQUENT
STRAJUSUM

JavaScript Foundations

COERSION RULES

- ✦ undefined = false
- ✦ null = false
- ✦ 0 = false
- ✦ any other number (including negative numbers) = true
- ✦ "" = false
- ✦ "any other string" = true

AQUENT
STRAJUSUM

JavaScript Foundations

DON'T GET FOOLED!

```
if("false") {  
  console.log("false" is a non-empty string, so it's true!);  
}
```

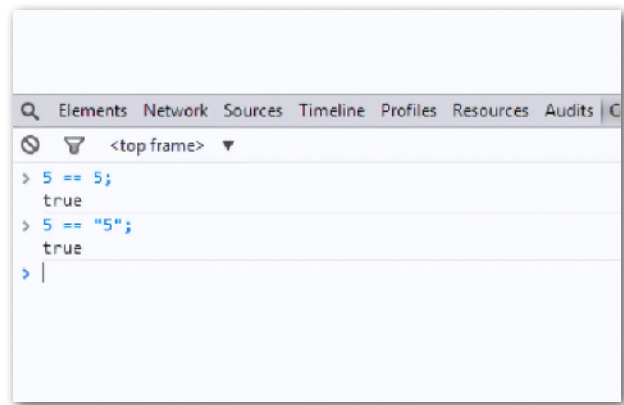
AQUENT
STRAJUSUM

JavaScript Foundations

false. Any other number is coerced into true, including negative numbers. Any number other than zero is true.

Strings are just as easy. An empty string would be two quotes with nothing in between them. This is coerced into false. Any other string is coerced into true.

A tricky one is the string “false”. This feels like it should become false, but it’s a non-empty string, so it does, in fact, get coerced into true.



```
> 5 == 5;
true
> 5 == "5";
true
> |
```

And, there’s one more connection I want to show you about comparisons and coercion. You’ve already used the double equals operator to check if two values are equal, like so. Five equals five. This, of course, is true. But try five equals the string “five”. You also get true. So, here we have a number being called equal to a string.

They are two different types of data, but JavaScript will try to coerce that string into a number. Because the string only contains a number, it does successfully coerce it into the number five, which is equal to the original five.

Now, generally speaking, if you have two different data types, you wouldn’t really expect them to be equal. This is considered by many to be an odd quirk in JavaScript. Some might even call it a bug. To get around this, there’s another comparison operator, the triple equal sign, or strict comparison operator.

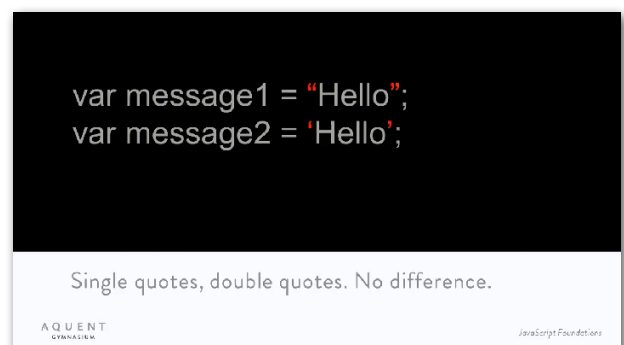
When you use this, it does not do any coercion. So, five equals equals equals the string “five”, results in false.

The double equals comparison has additional odd behaviors when comparing other data types, so most developers considered it a best practice to always use the strict version. I’ll be doing that for the rest of this course, so if you see a triple equals, you’ll know what’s going on.

Okay, that’s a lot to take in, so take a break, try it with some other examples in the console, or in a JavaScript file. And, when you’re ready, come back, and we’ll learn all about strings.

STRINGS

Strings are another one of the most fundamental data types, and also one of the most useful. As strings can hold virtually any textual data, you’ll find yourself using them constantly. You’ll use them to hold names, IDs, URLs, and even more abstract kinds of data. Now, as I said in the previous video, strings are defined by enclosing any text in single or double quotes. There’s no difference between these two.



```
var message1 = "Hello";
var message2 = 'Hello';
```

Single quotes, double quotes. No difference.

AQUENT CREATIONS JavaScript Foundations

Of course, you need to end the string with the same type of quote that you began it with. But, there is one instance where the difference between these quotes becomes very important. Say, you wanted to use single or

double quotes within your string. For example, say, you wanted to make string with the line, Joe says, quote, “Hello.” Well, let’s jump over to the console, and try wrapping that in double quotes. You do that, and you get a syntax error, unexpected identifier.

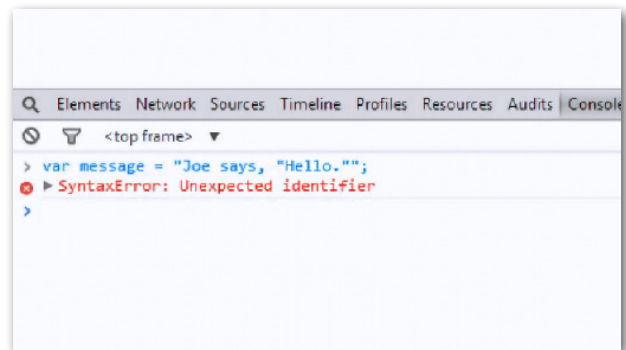
What happened is that, when JavaScript saw the opening double quote, it started to define a string. Then, it ran into the next double quote, just before hello, and took that to mean, the end of the string. Then, it sees this word, hello just sitting there, not in any quotes, and it’s not the name of any JavaScript keyword or variable, so, it reported an error. So, there are two solutions, here. One is that, if you need to use any double quotes in your string, define the string with single quotes, like so. And, that works fine.

Likewise, if you need to use single quotes in your string, define it with double quotes. This also goes for instances when you use a single quote as an apostrophe. Trying to do that with single quotes, the apostrophe, and the word “Joe’s” signals the end of the string. The solution is to use double quotes to wrap this string.

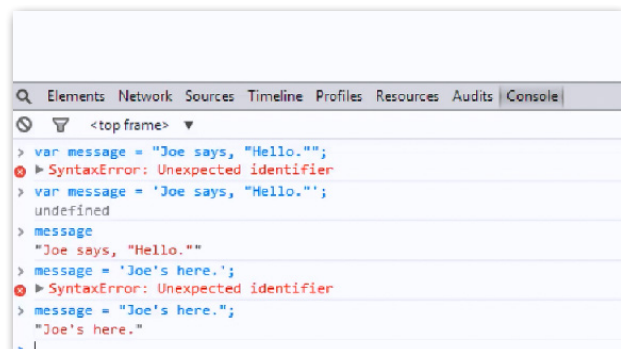
Now, there’s another solution, which is to escape the quotes. You escape a character by putting a backslash in front of it. The backslash tells JavaScript not to process the next character in the usual way. It escapes the usual treatment as something that starts and ends a string, and just gets seen as another character. If your string contains single and double quotes, this is the only solution. For example, if my string is Joe’s message is “Hello,” here, I can’t enclose the string in single or double quotes, because it contains both. So, what you do is choose one, and escape the ones you chose in the string. Let’s try that on console.

I’ll choose double quotes to make the string, which means I’ll need to escape the double quotes around the word “hello”, and that works. You can escape other characters as well. Some escape characters have special meanings. For example, backslash n is the new line character. So, if you type one line, backslash n, another line, that will output as if you pressed the Return or Enter key at that point in the string. Another common escape character is the backslash, itself. Say, you need to display a Windows file path, which is specified with backslashes. Well, if you type something like this, “windows backslash files,” it prints “windowsiles.”

What it’s doing is trying to escape the f character, which actually defines a form feed signal. And, this is meaningless in the console, so, it just gets ignored. But, now your f is gone. So, what you do is, escape the escape. The first backslash is the escape character, which allows the second backslash to be treated as simple text, escaping its normal functionality.



```
> var message = "Joe says, "Hello."";
SyntaxError: Unexpected identifier
>
```



```
> var message = "Joe says, "Hello."";
SyntaxError: Unexpected identifier
> var message = 'Joe says, "Hello."";
SyntaxError: Unexpected identifier
> message
"Joe says, "Hello.""
> message = 'Joe's here.';
SyntaxError: Unexpected identifier
> message = "Joe's here.";
"Joe's here."
>
```

```
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> message = 'Joe's here.';
✖ SyntaxError: Unexpected identifier
> message = "Joe's here.";
"Joe's here."
> var message = "Joe's message is \"Hello\".";
undefined
> message
"Joe's message is \"Hello\"."
> "one line\nanother line"
"one line
another line"
> "windows\files"
"windowsfiles"
> |
```

Okay, now, you know more than enough about how to create strings. What can you do with them? Well, you can add smaller strings together to make bigger strings. This is called, concatenation. You can use the plus operator to add strings together. Note the space after “Hello.” Concatenating strings adds the characters together, exactly as you write them. It doesn’t add anything. So, if you want a space between the two words you’re concatenating, make sure to include a space.

Let’s try this in the console. I’ll make a string in one variable, and another string in another variable, and then, concatenate them. Now, remember that with numbers, there are various operators that also assign, such as plus equal for adding a sign, minus equal for subtracting a sign, et cetera. While most of these don’t make much sense for strings, but plus equals does. It basically appends a string onto an existing string. So, if I say language equals Java, and then, language plus equals script, language has now become JavaScript.

Now, one more word on concatenation; you can also take advantage of coercion, while concatenating. Say, you have a function that generates a user number, and you want to combine that with a string to make a username. If I define num as 1234, and name as user, and username as name plus num, JavaScript wants to concatenate a string to a string, but it sees a number, instead. So, it coerces that number into a string so that it can concatenate it. So, now you can use concatenate to create long strings from short ones.

What about the opposite; subtracting shorter strings from longer ones? JavaScript actually has two functions for extracting substrings. In possibly the worst

```
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> "Java"
"Java"
> language += "Script";
"JavaScript"
> language
"JavaScript"
> var num = 1234;
undefined
> var name = "user";
undefined
> var username = name + num;
undefined
> username
"user1234"
> |
```

Getting strings from within strings:
substring()
substr()

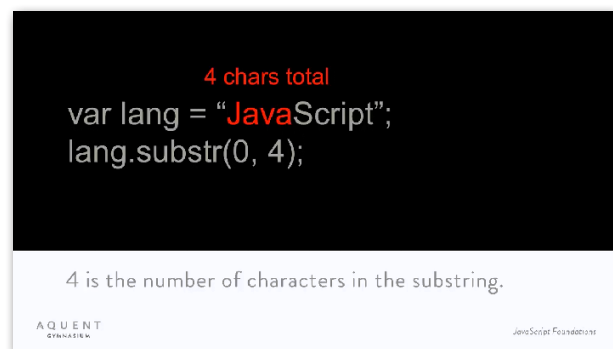
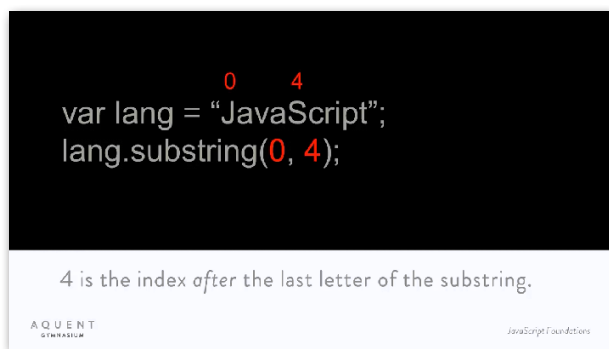
Seriously, who named these functions?

AQUENT
CORPORATION

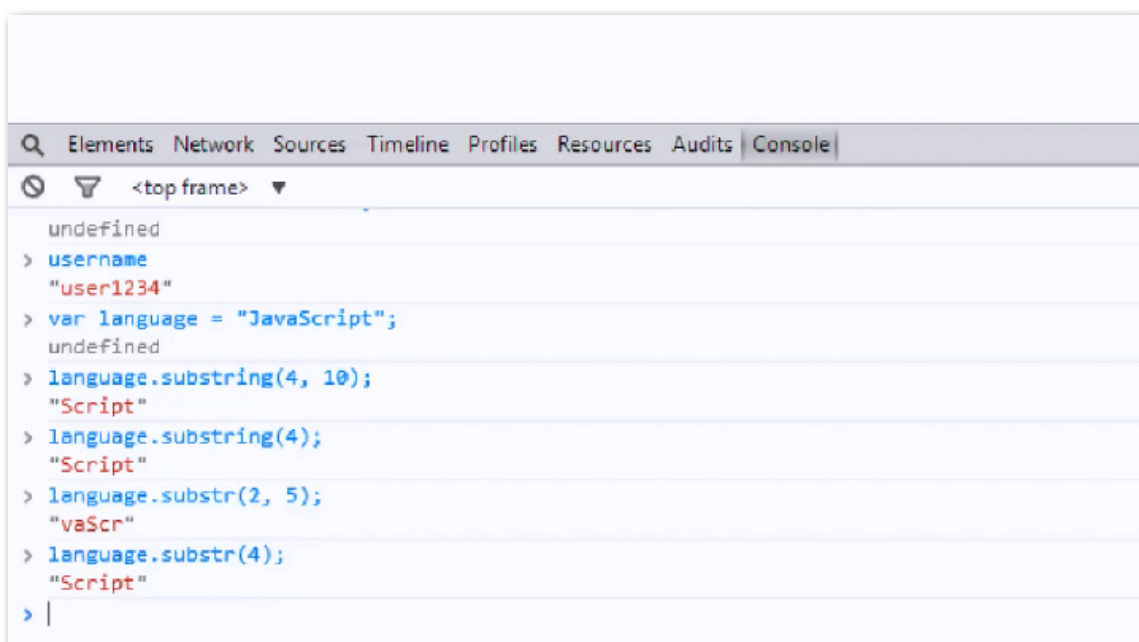
JavaScript Foundations

naming scheme ever, they're called substring and substr. They do exactly the same thing, but in slightly different ways. Both substring and substr let you extract a sequence of characters from within a string. Substring lets you pass in two numbers. The first number is the position of the first character in the substring you want. You should know that string characters are numbered starting with zero. The second number is the position of the first character that you don't want in your substring. Or, to put it another way, it points to the position after the last character you want in your substring.

I know that's a bit confusing. Anyway, here you can see language substring (zero, four). This starts with character zero, the first character, which is j, and ends just before character four, which is s. So, you get Java. Let's try another one, live. I'll define language as JavaScript, then, say, language substring (four, 10). This starts with character four, s, and ends just before character 10. Now, here, there is no character 10. But, this gives you a string up to character nine, which is t. So, you get script. Actually if you want a substring from any character to the last character in the string, you can just leave off the second number. So, language substring four also gives you script.



Now, substr does the same thing. The first number is still the first number of the substring. But, the second number specifies how many characters you want to get back. So, a language substring (zero, four) gives you Java. It starts at zero, and returns a string with four characters. Let's try that out with some other numbers. Language substring (two, five) gives you a five-character string, starting with the third character, v.



And, if you leave out the second number, `substr` works exactly the same as `substring`, giving you a string from whatever character you specified to the end. There are other little details about the way these two functions work, but I'll leave you to explore that on your own.

Another useful function, if you just want a single character out of a string, is the `charAt` function. Just pass a number to it, and it will give you the character at that position in the string. Let's try some of those in the console. Zero is `j`, one is `a`, two is `v`, et cetera.

Finally, say, you want to know whether or not a string contains a specific substring, or a particular character. That's what `indexOf` was for. Pass in the string or character that you're looking for, and it will return the position where that string starts. For example, here a language `indexOf` of `rip` returns six. This tells us that starting at position six, you'll find the word "rip" in JavaScript. If the substring a character does not exist in the larger string, this will return minus one.

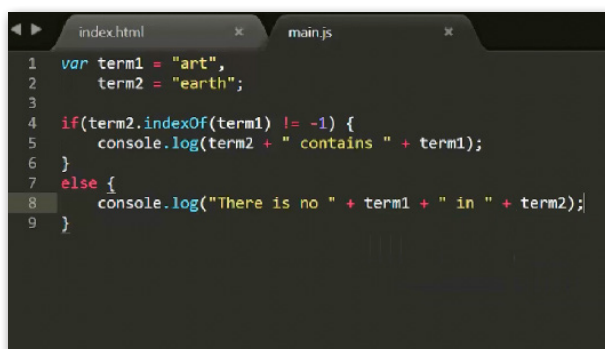
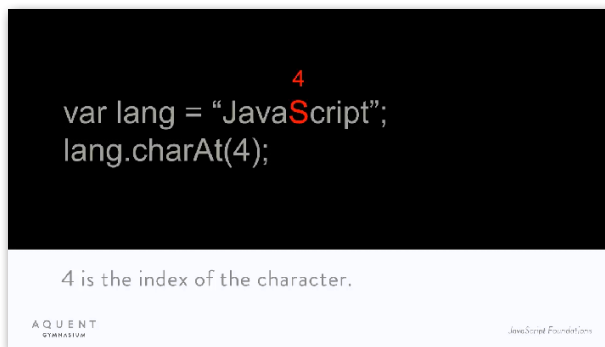
Armed with this knowledge, we can prove once and for all that there is no `l` in `team`.

Now, you can use `indexOf` with not equals minus one, to see whether or not a string contains another string. I'll create `term one` and `term two` variables that contain strings. And, I'll check `term two` `indexOf` of `term one`. If that returns any number other than minus one, then `term one` exists in `term two`. In other words, if the result is not equal minus one, then I found a match. I'll just log the results in the appropriate `if` or `else` block, and test that. Then, I can change `term one` and now `indexOf` should return minus one. And, the `else` code block will execute, instead. And, that's exactly what happens. Now, there are lots of other functions that you can perform on strings. Again, I don't expect you to completely memorize everything I've covered here, but you should come away with a solid knowledge of what strings are, and a pretty good idea of some of the ways you can manipulate them.

In lesson two, you'll see how to create your own functions to add custom actions and behaviors. And, I'll introduce you to some more advanced data types that will allow you to store collections, or lists of values, and create your own data types that are composed of various values. At the end of the next lesson, you'll be able to write some real programs, instead of the small snippets you've been restricted to so far.

But, before moving on, you have a few assignments to do. Assignment one is the quiz that you'll find on the page for this lesson. There will be a quiz for each lesson. This will help you see how well you understood the lesson, and maybe, point out some areas to go over, again.

Assignment two, go to a website of your choice, open the developer tools to the Sources tab, find some JavaScript files, look for some variables, strings, numbers, and an `if` statement. Don't worry if you don't under-



stand what the code is doing. Just try to see what you can recognize. Sometimes the source will be minified. This means that all the extra spacing and formatting has been removed to make it smaller and harder to read. At the bottom of the Sources tab, you'll see a button, like this. This will attempt to format the code to make it more readable for you.

Assignment number three, check the console on a number of web pages. Do you see any errors or warnings? Can you find any log statements that a developer might have left in the code?

Assignment four, choose a code editor that you'll use for the rest of this course. Make sure it's installed, and that you're familiar with how to use it. Unless you have one that you're already very familiar and comfortable with, I suggest Sublime Text. Create a project that has an HTML file that loads a script file. Have that script file successfully log a message to the console.

Assignment five, pick some topic that was covered in lesson one that you didn't completely understand, or would like to know more about. Do some personal research to understand that topic more fully. I highly suggest the Mozilla developer network for this purpose.

ASSIGNMENT #2:

Go to a web site of your choice. Open the developer tools to the sources tab and find some JavaScript files.

Look for some variables, strings, numbers and an if statement.

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #3:

Check the console on a number of web pages. Do you see any errors or warnings? Can you find any log statements that a developer might have left in the code?

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #4:

Choose a code editor that you will use for the rest of this course. Make sure it is installed and that you are familiar with how to use it.

I suggest Sublime Text (www.sublimetext.com).

Create a project that has an HTML file that loads a script file. Have the script file successfully log a message to the console.

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #5:

Pick some topic that was covered in Lesson 1 that you either didn't completely understand, or would like to know more details about.

Do some personal research to understand that topic more fully. I suggest checking the Mozilla Developer Network:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

AQUENT
GYMNASIUM

JavaScript Foundations