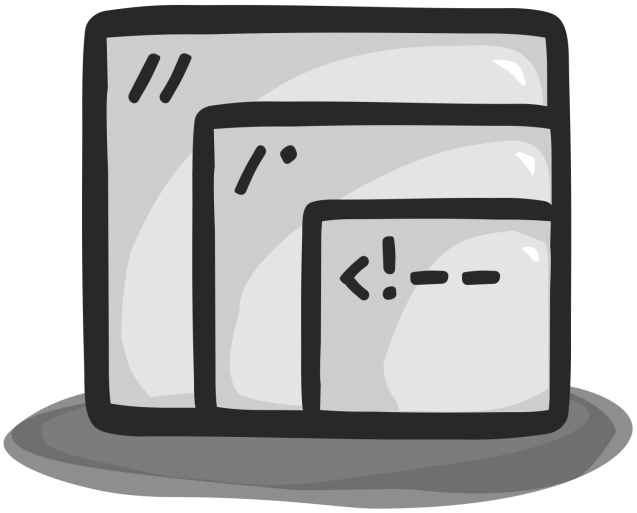


GYMNASIUM

INTRODUCTION TO MODERN WEB DESIGN

Lesson 6 Handout

Getting Started With JavaScript

ABOUT THIS HANDOUT

This handout includes the following:

- A list of the core concepts covered in this lesson.
- The assignment(s) for this lesson.
- A list of readings and resources for this lesson including books, articles and websites mentioned in the videos by the instructor, plus bonus readings and resources hand-picked by the instructor.
- A transcript of the lecture videos for this lesson

CORE CONCEPTS

Note: this lesson provides a brief overview of JavaScript, for a deeper dive into the language consider taking the Gymnasium course "JavaScript Foundations". (https://thegymnasium.com/courses/GYM/104/0/about)

1. JavaScript is a scripting or programming language that adds interactivity to web pages, specifically it allows us to find and manipulate elements and content on our web pages. We can do things like validate forms, perform calculations, look up information about the browser, we can also communicate via JavaScript directly with the server, and do a lot more.
2. A good front-end developer will make sure that their web pages are not 100% dependent on JavaScript for delivering content. In extreme cases, a poorly designed page that relies on JavaScript may cause a site to not render at all. In many cases, poor use of JavaScript can also completely reduce the accessibility of your site or exhaust the system resources in a user's browser including RAM and/or bandwidth.
3. Most modern browsers now include a JavaScript console within the browser developer tools, this console is extremely useful for experimenting with JavaScript code and for debugging issues with your code.
4. JavaScript allows you to work with a number of different data types, such as numbers and strings. Other common data types that are available include arrays, (which are collections of other values) objects, and Boolean values, (which are true or false.)
5. Variables are a key starting point for working within JavaScript. A variable is something a developer defines with a name and allows you create a placeholder for one or more data types and then reuse this variable as many times as she likes.
6. One of JavaScript's features is the ability to locate HTML elements within a document, at a later point you will be able to manipulate these elements, but the first step is finding them. For example, you can use the method called getElementsByTagName() in order to search for specific elements by their tag name within your document.

7. Another important feature of JavaScript is the ability to insert information or content into an existing web page. In the example of this lesson we locate a specific class in our HTML document, extract the content from that class and then pass it along to another location.
8. JavaScript has the ability to respond to the way a user interacts with the web page. These are generically known as event listeners, an “event” from JavaScript’s perspective is something the user does such as tapping, clicking and dragging (there are additional events, but these are extremely common.)
9. A JavaScript event has two phases: a capturing phase and a bubbling phase. The capturing phase can be thought of as all the elements in the DOM tree that are registered by the event. The bubbling phase begins immediately after an event reaches the end of the Dom tree and begins to moves back up triggering whatever code has been assigned.
10. On mobile touchscreen devices, the click event is often delayed by 300 milliseconds. in order to allow for somebody to potentially double tap. However, there are other touch-based events, such as touchdown, that do not have that 300 millisecond delay. So in order to accommodate both a touch screen and a mouse-based interface, be sure to test your pages when using events and modify your code as needed to adapt.
11. You can easily include third-party JavaScript code into your projects, typically by including a third-party reference to the JavaScript code. This can save a tremendous amount of time and allow you to reuse code that has been tested by others in real-world scenarios. As with all third-party code, be sure you understand what you are adding and do not add code without doing research and testing.
12. Refactoring code is a term used for the process of optimizing your JavaScript for functionality and performance. This is typically accomplished by reviewing your code and then editing it as needed. Examples of refactoring might be reviewing your code to identify events that are being fired unnecessarily or looking for needlessly repetitive code.

ASSIGNMENTS

1. Get practice traversing your document using JavaScript. Specifically, open one of your pages in Chrome and then open the JavaScript console. Use some of the methods covered in Chapter Two to select different elements on the page. Consider the different options and reflect on what’s selected by each method, how easy they are to use, and which one might be most appropriate in different scenarios. Post your observations and thoughts in the forum.
2. Experiment with manipulating elements on your page. Specifically, open up Chrome and the JavaScript console and then inspect an element in the Chrome Dev Tools using the DOM tool and look at the other properties that you can read or write on the element. Try getting and setting them via the JavaScript console. Remember, get attribute and set attribute. Post your observations and thoughts in the forum.
3. Make sure you understand how to work with events on your page. Specifically, open up Chrome and the JavaScript console, and use addEventListener to assign a click handler to the HTML element, the body element, and a paragraph element that triggers console.log(this.nodeName). Make sure the third argument is false. Now what happens when you click your mouse on that paragraph?

- Now assign a second set of the same event handlers with the third argument set to true. What happens when you click now? Post your findings in the forum.

4. Post your project to GitHub. At this point, you should have a complete site that includes the HTML and CSS concepts you've learned, as well as the JavaScript navigation menu created in this lesson. Once you're done, post a link to the site in the forum so your classmates and the Gymnasium teaching assistants can give you feedback.
5. Take the final exam. Be sure to study up on the Quiz questions from lessons 1-6, you will get two chances to pass the exam. If you do pass, you'll get a signed certificate from the instructor. This certificate can be shared in a number of different ways, including LinkedIn, so you can show everyone the hard work you put into this course!

INTRODUCTION

(Note: This is an edited transcript of the Modern Web Design lecture videos. Some students work better with written material than by watching videos alone, so we're offering this 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.)

Now this lesson is going to be a little bit different, a little more focused on practical application of technology, rather than a survey of what's possible. It will be a little bit of theory and a little bit of practice. So the general idea for this lesson is to focus it around a project-- specifically, converting the navigation of my sample site from a dropdown to a drawer nav on small screens.

The lesson files that I'll be going through are on the class site and available to you. I just wanted to take a moment to introduce you to the idea that we'll be walking through. We'll get into the details a little bit later on.

So kicking things off, what exactly is JavaScript? JavaScript is the focus of this lesson. So what is it? What can it do?

Well first of all, like anything awesome, it's got a logo. But when we're talking about JavaScript, what we're able to do with it as a programming language is we're able to find and manipulate elements and content on our web pages. We can do things like validate forms, perform calculations, look up information about the browser, we can communicate via JavaScript directly with the server, and do a lot of other things.

As this is not a JavaScript course but simply a lesson introducing you to JavaScript in the context of modern web design, I wanted to give you an introduction to the overall idea of JavaScript and what you can do with it. For more, you should definitely take a look at the JavaScript Foundations course. This lesson is limited by time, whereas that course is a deep dive into the world of JavaScript.

So JavaScript can do all of these awesome things. But it can also do some pretty heinous things as well, such as completely hose your site. If you're 100% dependent on JavaScript for delivering your content and something goes wrong, the site may not render at all. You can also completely reduce the accessibility of your site to people who may be using assistive technologies. You can also do things like simply exhaust all of the system resources that your user has available to them. That might be RAM or it might be the amount of bandwidth they have, if they're on a pay per bit connection.

So JavaScript is a very, very powerful programming language. But as Uncle Ben from Spiderman tells Peter Parker, "With great power, there must also come great responsibility." So we need to be very careful about how we use JavaScript. And throughout the course of this lesson, I'll be going through some basic ways to use Ja-

vaScript. And then in the final chapter, I'll actually be going through and showing you how to refactor JavaScript to be a little bit nicer, and perform a little bit better. So definitely pay attention to that final lesson.

OK, to kick things off I just want to demonstrate to you that you can actually run most of the code examples that I'm going to be showing you in the course of this lesson directly within the JavaScript command line in Chrome, or what's called the Console. Here you see me firing up the Console, which you can get to simply by hitting Inspect Element. Or, if you're on a Mac, Command Option J.

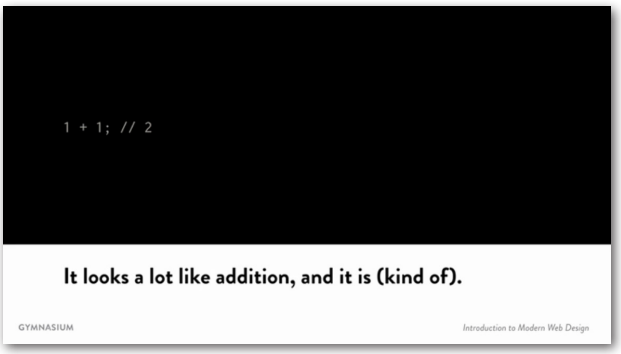
And then in that, you can type most of the different JavaScript calculations and code directly in there that you want to interact with. So you can do things like math functions. I can go ahead and write in some strings and concatenate those.

Don't worry about following along with what I'm typing in here right now. I'm just showing you what the Console is and how you can go ahead and input code directly in there, just to familiarize you with that. But I'm going to walk through exactly what JavaScript is and how it works right now.

So JavaScript looks something like this. This is a very, very simple statement. So statements are instructions to the JavaScript processor that end with a semi-colon. In this case, the statement here is concatenating two text strings.

So the text strings are denoted with the single quotes there on either side of the word. And the plus symbol between them is concatenating the two, and it's making the hello and world run together. So here you see the output of that in a comment on the end of the line.

So we seem to be missing a space, because when we're concatenating these, neither of them has a space in them. We can get past this issue by adding the space to hello, or adding it to world, or we could add a third string in the middle that just contains the space. So this gives you kind of a simple example of JavaScript statements and concatenation, and how that works.



Concatenation looks a lot like addition. And it is kind of like addition. In JavaScript we have the ability to say 1 plus 1 and get the result of 2. We can also say 1 minus 1 and get 0, 1 times 1 and get 1, 1 divided by 1, still get 1. We can do lots of different math functions.

But in the world of concatenation, instead of adding numbers together, we're actually adding strings of text together. So hello plus space plus world gives us hello space world. Strings are text values and they can come in a variety of forms.

Sometimes you'll see them with double quotes, sometimes you'll see them with single quotes.

The most important thing to keep in mind, however, is that if you use double quotes to wrap your strings and you need to include a double quote character inside of that string, you need to escape it with a backslash. The same thing goes for the single quotes. You can see that in the second example here, and the fourth example here.

Generally I prefer to do my strings in single quotes. And the reason that I really prefer single quotes is that I'm often generating HTML markup in my JavaScript. And if I use single quotes, I don't have to worry about using backslashes to escape the double quotes when I'm using an attribute.

```
'hello' + ' ' + 'world'; // hello world
```

Instead of adding numbers together you're adding strings together.

GYMNASIUM *Introduction to Modern Web Design*

So JavaScript can do some pretty interesting things, like add a number and a string together. So what happens in this situation is that the two become concatenated, because you can't add world to 1. So what happens is that JavaScript coerces the number into a string, and then concatenates the two text strings together. So you end up with 1world.

```
1 + 'world'; // 1world
```

The number is coerced to a string, so the two can be concatenated.

GYMNASIUM *Introduction to Modern Web Design*

JavaScript is sometimes referred to as a loosely typed language, or duck typed. Because if it walks like a duck and talks like a duck, it must be a duck. But it's referred to as a loosely typed language because it allows values to be coerced like you're seeing here.

In strictly typed languages the 1 would only be able to be a number, and could not be concatenated with a string. You would have to convert it specifically to a string in order to do that concatenation. Doing something like this would throw an error. But JavaScript allows us to coerce a value from a number to a string, and so on and so forth.

Now numbers and strings are only some of the different data types that we have available to us in JavaScript. Other common ones include arrays, which are collections of other values. So in this case, a collection of four numbers. We have objects, which we'll get into a little bit later. We also have Boolean values, which are true or false. And it's important to note that numbers can be negative. They can also have decimal points, and so on and so forth.

```
-1;                  // Number
3.14;                // Number
[ 1, 2, 3, 4 ];      // Array
{ property: 'value' } // Object
true;                // Boolean (see also: false)
```

JavaScript offers a variety of other data types.

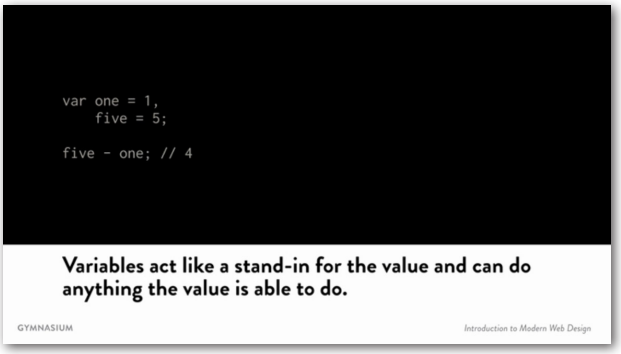
Throughout the course of this lesson we'll see a wide variety of different data types. Now in the examples you've seen so far, I've just been adding data types together, right? We just concatenated the strings, writing them literally into the statement. But that's not terribly efficient if we need to use that string multiple times.

For that reason, we may want to do something like store that string within a variable. You can think of variables as buckets where you're placing some data. So in this case, I'm using the var declaration to declare my variable hi is equal to the text string hello. So that's what's referred to as a var statement. I can also say var hi equals hello, and then make another var statement, there equals world.

JavaScript also allows us to shorten this up a little bit by using a comma to create a compound var statement. So in this case, I'm assigning hi the value of hello and there the value of world, just as I had on the previous screen. But in this case, I'm doing it without having to do separate var statements. I can just chain all of them together in a single var statement.

Once I have these values stored in the variables, I can go ahead and use them as replacements for those values. So if I was to say hi plus there, you would get helloworld without the space, like we saw earlier. Now interestingly enough, I could still do that same concatenation with the empty space string in the middle in between the two variables, and I get exactly the same result as I would have had I done it just with the string literals.

And you can assign any sort of data type to these variables. So I could create a variable one, as in the word one, is equal to the number 1. And five is equal to the number 5. And then if I say five minus one, I'll get the value 4. So variables are simply placeholders for the values that they contain.



Now when you're creating variables, you can name them pretty much anything you want. Generally, it's a good idea to choose variable names that actually make sense and provide a little bit of information about the type of content that they contain. So rather than saying text equals check availability, if I'm using it in the context of a button, maybe I want to say button underscore text equals check availability, because that's going to be the text that's going to show on the button. It makes it really easy for me to look at that variable name and know exactly what it contains.

Good variable names are also easy to read. I personally prefer using lowercase with an underscore between words. But you could do capitalization. You could also run words together.

You could capitalize the individual words and run them together. You could do CamelCase, which is the final example there. There's a lot of variety.

Pick one that's easy to read and stick to it. Be consistent. So don't have all of those different ways of writing your variable names. Pick the strategy that you like and stick to it.

```
var button_text = 'Check Availability';
var $button = document.createElement( 'button' );
```

Good variable names should be understandable at a glance.

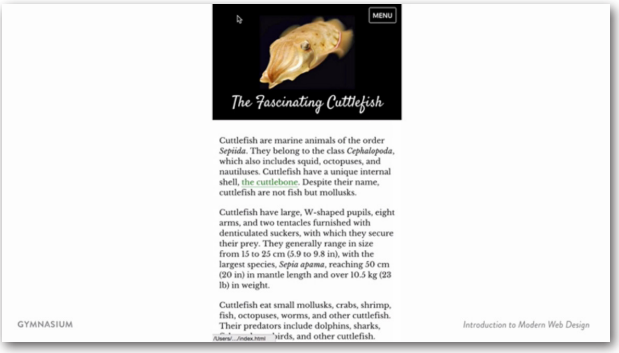
So in this case I've got button underscore text, and item underscore count. Two different variables that do two different things, but the naming is consistent. Another hallmark of good variable names is that they're understandable at a glance.

For this reason, I like to use signifiers like the dollar sign, in this instance, to indicate that this particular variable is a reference to a DOM Node. So it's a reference to an element within the HTML page or an element that I'm creating within my program. It makes it really easy to distinguish one variable from another when I'm scanning through a rather large script.

So as I mentioned earlier, the idea, the project that we're going to go through in the process of this lesson, is to convert my navigation from a dropdown to a drawer on small screens. OK? So it'll look a little bit something like this. If I click on the menu, it will slide in this drawer navigation. When I click outside, it will disappear again.

Now the reason that I want to do this interaction is that the default small screen interaction is to have the navigation at the end, and then to jump to it when you click the menu icon. I don't necessarily want everybody to have to do that. So I can use JavaScript to make this simple enhancement that will create this drawer nav. We can do a lot with CSS and HTML, but this particular effect requires tweaking styles applied to elements further up in the DOM tree, which we can't currently do with CSS alone. We need to use JavaScript.

Now each chapter in this lesson builds on the previous one. So it's important to give yourself the time and the headspace to focus on the content. Otherwise you could easily get lost. The whole lesson will take about an hour to complete.



So the plan for building this interface is that we need to check and see if the screen is small. And if it is, we're going to add a class to the HTML element that flips on the styles that will make the drawer work. The reason that we're doing this is that we don't want those styles applied if the drawer is not capable of actually running. We want JavaScript to essentially flip a switch that will turn on those styles.

If we didn't do that, if we set the styles to be on by default, somebody who's on an older browser or a browser that experiences an error with our JavaScript program all of a sudden may not be able to access the navigation menu at all, because we're going to be hiding it using CSS by default. And without JavaScript, they wouldn't be able to expose it. So once the styles are turned on, the next step is going to be to make the menu button open up the Navigation drawer by adding a new class to the HTML element. And then finally, we need to make clicking outside of the menu close the Navigation drawer by removing that class from the HTML.

To set things up, I'm going to go ahead and create a new file named main.js, and I'm going to put it in a new directory in my site called js. And then I'm going to include it using the script element, as you see here, just at the end of the body. So script source equals js slash main.js. And then I'm also going to set it to be asynchronous using the async attribute. And what that does is it informs the browser to load this script, but to not allow it to block rendering while it's downloaded and parsed.

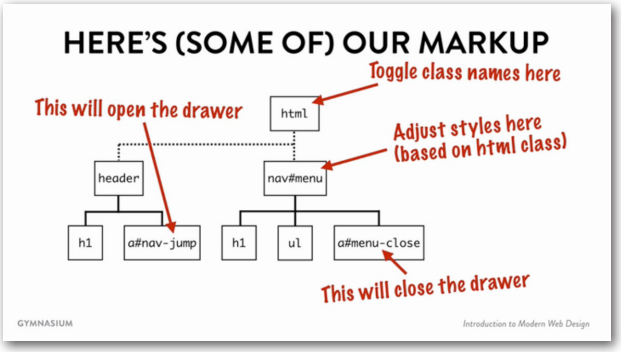
This sets everything up for me to begin writing the scripts that we'll discuss in the subsequent chapters. So in this chapter, we've discussed what JavaScript is, what it can do, using the JavaScript Console in Chrome, statements in JavaScript, and variables. In the next chapter, we're going to look at extracting information from a web page using JavaScript. I'll see you in a few minutes.

FINDING THINGS

Welcome back to chapter two, where we're going to look at finding things in our HTML pages. I'm going to give you a very basic introduction to wayfinding within an HTML document. And you can learn more about these approaches in lesson three of the JavaScript Foundations course.

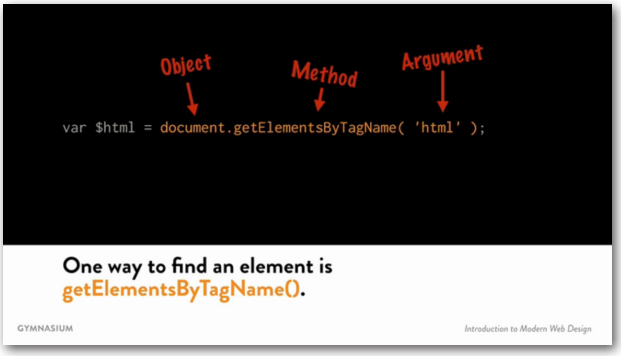
Let's start by looking back at my plan. In my plan, I make reference to several elements. Remember when we talked about the DOM in the HTML in the CSS chapters? Well, this is a representation of the document that we can traverse using JavaScript to find the elements we want to do things with. So here is an excerpt of the markup from my site. I have the HTML element. And then I've got a header, which includes the h1 and A at anchor with an ID of nav-jump. Then I have the actual menu later in the markup. And that is nav with an ID of menu. And then inside of that nav, I've got an h1, an unordered list, and then a with an ID of menu-close.

So in terms of the plan, in the elements that we're working with, this first link will open the drawer. This final link will close the drawer. And the HTML element is where we're going to toggle the class names. Most of the actual style changes, however, will be taking place on nav with an ID of menu.



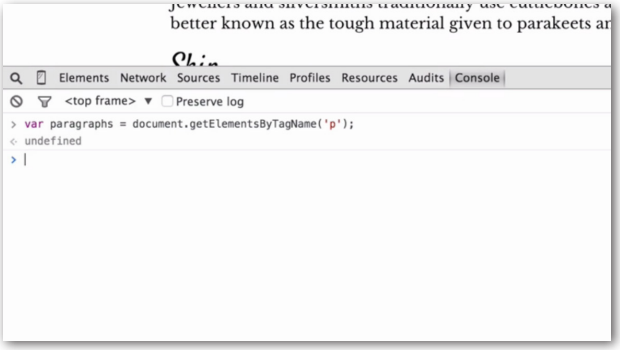
So one common way to find elements within a document is to use document.getElementsByTagName. The bit inside the parentheses there is what's called an argument. And the argument in this case is the tag name that we want to find. Now breaking this down a little bit, the first bit we have there before the period is the object that we're dealing with-- in this case, the document. The document is a universal object in the document object model and refers to

the page as a whole.



That object has a method called getElementsByTagName that is available to us to search for elements by their tag name within the document. And then finally, the last little bit inside those parentheses there is the argument-- in this case, the string HTML. Now, it's important that you pay attention to the fact that this is elements, get elements plural by tag name. So what it returns is a collection of elements that have the tag name HTML.

A collection is sort of like an array. And in order to get access to the actual HTML element and not that collection, we need to pluck that particular element from the collection using square bracket syntax. This may seem a little bit weird. But what I'm saying here is that I want to pluck the first element of that collection and assign it to $html.



Now, in order to demonstrate this a little bit better for you, let's hop over to the browser. And in the browser, if I go to the console and I create a variable called paragraphs, and then I say, document.getElementsByTagName, and then I put the p in there as my string argument, it collects all of the paragraphs from the page into the paragraphs variable. And there, you can see them coming back as a collection.

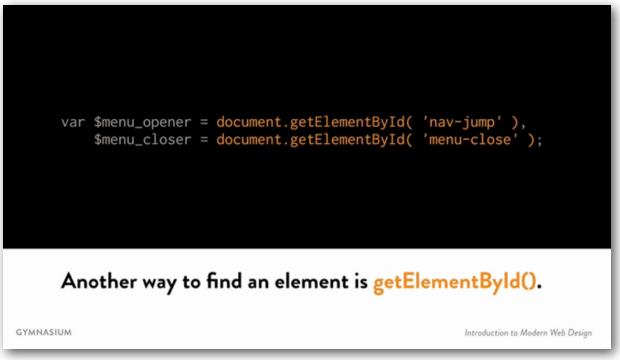
Now, if I want to get the first one, I can say paragraphs. Oops. I had a little typo in my script there. Pluck the first one, paragraphs, and then zero in square brackets. And you can see, there is the first paragraph. If I wanted to do paragraph 13, which would actually be the 14th, since it's zero-based array collection, it would actually get me the 14th paragraph, which is underneath suckers. So that's pretty cool. And that is how getElementsByTagName works.

Now, another way to find elements within the page is getElementById. Now, you'll remember I said that the ID attribute needed to be unique to the page when you assign one to an element. And that's because it is a reference point for getElementById. And it's also an anchor point within the document, as we saw in the target example in the CSS lesson.

Now, document.getElementById by ID will search for the ID value nav-jump in the first instance and menu-close as values for that ID attribute within the context of the entire document. It's important to pay attention to the fact that it's element singular, not elements plural. And again, that comes back to IDs being unique on the page.

Hopping over to the browser again, if I create a variable, var element is equal to document.getElementById circulation, then if I refer to element, it is that circulation section right there. So that's pretty cool. Now, interestingly, once I have that element, I can use that as the context for getElementsByTagName. So I could say element.

GetElementsByTagName h1 and then square bracket zero, and it will pluck that first h1 for me. So that's pretty cool. It makes it easy to traverse documents and to move into nested markup structures.



Now, we can combine all of these statements into a single var statement. So I have html=document.getElementsBy-TagName html square bracket zero, menu opener=document. GetElementById nav-jump, menu closer=document.GetEle-mentById menu-close.

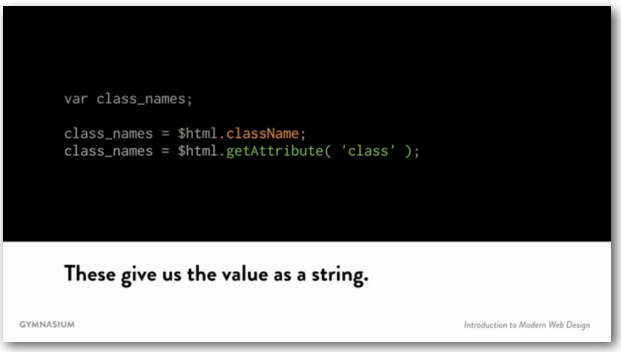
All right. So that's a few different ways to traverse the doc-ument and move around within it. I want you to play around with these different techniques in assignment one. You can do that by opening up one of your pages in Chrome and then opening the JavaScript console. And then use some of the methods I have covered here to select different elements on the page. Consider the different options and reflect on what's selected by each method, how easy they are to use, and which one might be most appropriate in different scenarios. Post your observations and thoughts in the forum.

So in this chapter, we've discussed extracting information or identifying information within a web page. In the next chapter, we're going to look at inserting information into a web page. I'll see you in a few minutes.

MANIPULATING THINGS

Welcome back. In this chapter we're going to look at manipu-lating elements within our document. Specifically, we're going to look at manipulating the class attribute on the elements that we found in the last chapter. I'm going to give you a basic introduction to manipulating an HTML document. But you can learn more about this approach in lesson three of the JavaS-cript Foundations course.

So returning to our plan, there are three instances where I want to tweak the class assigned to the HTML element. Now we can access class values in a few different ways. If I define the variable class_names, that information can be extracted from the page using the class name attribute on the HTML element, or any element within an HTML document. Or, I could get it using .getAttribute and then pass in the attribute I'm looking for, in this case the text string class.



The first is an instance of the HTML Dom API. And the second is what's referred to as a getter method. The first instance can

be both read and written to. So if I were to say HTML.classname equals Foo, text string Foo, that would assign that value to the class, overwriting anything that existed previously. Get attribute, however, only gets the attribute value. It does not allow me to set that attribute value. To do that, I would have to use set attribute, which is a setter method.

Either of these approaches would get us the value of the class attribute as a string. Once we have that string, we can manipulate it in a number of different ways. Here's a simple example of concatenation. In this case, using the plus equals syntax.

```
class_names += ' my-new-class-name';

// or

class_names = class_names.split( ' ' );
class_names.push( 'my-new-class-name' );
class_names = class_names.join( ' ' );
```

Or we could convert the string to an array, append the class name to that, and then convert it back into a string.

GYMNASIUM *Introduction to Modern Web Design*

Now, what plus equals is equivalent to is saying class names is equal to class_names plus equals space my-new-class-name. I could write it out in that long form, or I could simply say class_names plus equals space my-new-class-name. SO that's one way to append the value to the class_names.

The other is that I could split the class value on a space. It says classes are space separated. And then I could push the new class name to that array. And then I could join that array back together again using a space. And that would convert it back to a string again. Either of these would work perfectly, but the second is just a little bit more surgical in how it does it.

```
class_names = class_names
                .replace( 'my-new-class-name', '' );

// or

class_names = class_names
                .replace( /\bmy-new-class-name\b/, '' );
```

We could remove the value using the replace() method with either a string or a "regular expression".

GYMNASIUM *Introduction to Modern Web Design*

Now removing the class name is a little trickier. I could use string replacement to replace my-new-class-name with an empty string. Or I could use a regular expression, in the second example here, to replace my-new-class-name with some sort of word boundary, which is either a space or a carriage return, where at the start or end of the overall string with an empty string. The first is fairly straightforward. The second is a little bit more complicated.

```
class_names = class_names.split( ' ' );

var len = class_names.length,
    kept_classes = [];
while ( len-- ) {
  if ( class_names[len] != 'my-new-class-name' ) {
    kept_classes.push( class_names[len] );
  }
}

class_names = kept_classes.join( ' ' );
```

This creates a new, empty array (which we will fill with the class names we want to keep).

But if we want to get surgical about how we remove class names, we have to get a little craftier. So let me walk you through this code here. First of all, we have our class names variable, which I'm going to split on a space so that I have an array of class names. And then going to capture the length of that array as the Len variable. And then I'm going to create a second variable called kept classes, which is going to be another array. It's going to be an empty array, which is what I'm saying there. Square brackets simply means empty array. And that array is what we're going to fill with the class names that we want to keep.

Finally, I can loop through those class names. In this case, I'm iterating over the class names, working my way from the end of the class list to the start of the class list. That's what that minus minus does. It's decrementing the value of the length each time the loop is run. So if this particular element was to have four class names, the

length would be four. Len would be equal to four.

So when I say, while len minus minus, that means that the length value would be decremented after that condition is evaluated. So within those curly braces, the gray text in between them, the value of len would actually be three because it had been reduced after it was evaluated. And, as I went over, arrays are zero indexed, which means that class_names square brackets len, as you see in that next line, would actually be that fourth class name.

It may seem a little bit weird, but once you wrap your head around zero indexed arrays, it makes complete sense. So the while statement will actually run until the content of the parentheses evaluates as false. That bit in green is what's called a test condition. So as the loop is going, when it gets that final class name, whose index is zero, and it goes back to run the loop one last time, zero would be inside of those parentheses. And zero is what we call a falsie value.

```
class_names = class_names.split( ' ' );

var len = class_names.length,
    kept_classes = [];
while ( len-- ) {
  if ( class_names[len] != 'my-new-class-name' ) {
    kept_classes.push( class_names[len] );
  }
}

class_names = kept_classes.join( ' ' );
```

The contents of a while statement will be executed until the test condition is false.

GYMNASIUM *Introduction to Modern Web Design*

So it would return as though it was false. So it would be seen as though it was false. And the loop would stop. So within that while loop, we're going to test a condition. In this case, we're going to test to see if the current class name-- so 4, 3, 2, 1, or simply 1, if there's only one class name-- is not equal to my-new-class-name.

So in this case, if it is equal to my-new-class-name, the gray text inside of the curly braces would be skipped over, would not be executed. But if the current class name was equal to my-new-class-name, the contents of the if conditional would not be evaluated. But if the class name is not equal to my-new-class-name, it would be evaluated, and that class name would be pushed to the kept classes array.

The exclamation point equals or bang equals operator is another comparison operator that we have to test to see if two things are equal. To find out more about operators, check out the JavaScript Foundations course. If that condition evaluates to true, the current class name is then pushed to the kept classes array.

Finally, we go ahead and rejoin the kept classes and assign those back to the class_names variable. Now that's an awful lot of code that we need to write each time we need to adjust the class name on a particular element. In order to make it a little bit more usable if we need to call it multiple times, as we do in the case of this particular script, we can abstract that to a function. In this case, I've created a function using the function declaration. And I've named it removeNewClassFromHTML.

```
function removeNewClassFromHTML() {
  var class_names = $html.className.split( ' ' ),
      len = class_names.length,
      kept_classes = [];
  while ( len-- ) {
    if ( class_names[len] != 'my-new-class-name' ) {
      kept_classes.push( class_names[len] );
    }
  }
  element.className = kept_classes.join( ' ' );
}
```

We could abstract that to a function.

GYMNASIUM *Introduction to Modern Web Design*

Now you'll notice that the way that I've written the names of these functions is a little bit different than the variables you've seen before. In this case, I'm using camel case, which means starting with a lower letter and then each subsequent word is run into the previous word, and it's capitalized. Now the reason that I like this approach is that it makes it really easy for me to look at a variable and tell that it's a function.

Additionally, I like to name them in a way that describes what they do. So in this case, removeNewClassFromHTML or addNewClassToHTML. Now it's worth noting that when you create a function, that function becomes a method of the global object, which is the window object. So if you'll note, I've created those two functions in the first two lines. And then if I were to in the console say removeNewClassFromHTML it would return that function to me.

```
var removeNewClassFromHTML = function(){};
var addNewClassToHTML = function(){};

window.removeNewClassFromHTML; // function(){}
window.addNewClassToHTML;      // function(){}
```

Functions and methods are technically variables, so naming them differently can help.

GYMNASIUM *Introduction to Modern Web Design*

So that's pretty good. But that particular function, remove-NewClassFromHTML, is pretty limited. It's just removing that class name from the HTML element. It would be better if I could create a more reusable function such as removeClass. Now in order to make it work, I need to define that function to have two arguments. The element that I'm talking about and the class name that I want to remove from that element.

With that set up in place, I can simply replace the references to the HTML element explicitly and that class name explicitly in order to achieve the exact same effect. And then I can simply say removeClass. $HTML is the first argument. And then the class name that I want to remove as the second argument. And that makes it really easy to use it anywhere in our code.

And then I could do the same thing for addClass. Now JavaScript is continuing to evolve, just like CSS and HTML are. And in newer versions of JavaScript and more modern browsers, we actually have a full API to do all of this stuff natively, and we don't even have to write those functions. It's a lot easier. And the way that it does this is via the classList API.

```
function addClass( element, class_name ) {
  var class_names = element.className.split( ' ' );
  class_names.push( class_name );
  element.className = class_names.join( ' ' );
}
```

We could do the same thing for adding a class.

GYMNASIUM *Introduction to Modern Web Design*

Now the classList is a collection of the class names assigned to that particular HTML element. And we have methods available to us to add, remove, toggle-- which means to add if it doesn't exist in the class list already and remove if it does-- and then we also have the ability to check and see if a particular class is contained within the classList.

```
var $html = document.getElementsByTagName('html')[0],
    page_classes = $html.classList,
    drawer_enabled_class = 'drawer-nav-enabled';

// if small screen
page_classes.add( drawer_enabled_class );

// if not small screen and currently enabled
page_classes.remove( drawer_enabled_class );
```

So, in my script, I can do something like this.

GYMNASIUM *Introduction to Modern Web Design*

So all of these make it super easy to do things like add and remove class names from an element. So if I were to say, here's the HTML element, $HTML is equal to document getElementsByTagName HTML square bracket zero. I can then create a variable page_classes, which is a reference to that classList. So page_classes is equal to HTML. classList. Then I could define a variable drawer_enabled_class that is equal to the class that I want to add and remove. Then I can simply say page_classes.add, drawer_enabled_class, page_classes.remove, drawer_enabled_class. Easy peasy.

Now because this is a newer API, we really should test to make sure that it's available before we use it. So I

could simply say, if classList is not in document body-- in other words, if the body element doesn't have a classList property, then I'm just going to return. Which means I'm just going to exit this program gracefully. I'm not going to throw any errors for the user, but they don't get the enhancement.

Now there's another instance where we'll need to use the classList API as well, and that's to add and remove the drawer opening class. For this next assignment, I'd like you to play around with manipulating elements on your page. So open up Chrome and open up the console and then inspect an element in the Chrome Dev Tools using the DOM tool and look at the other properties that you can read or write on the element. Try getting and setting them via the JavaScript console. Remember, get attribute and set attribute. Post your observations and thoughts in the forum.

In this chapter, we covered inserting information into a web page. Specifically, manipulating the class attribute. In the next chapter, we're going to look at listening for user interactions and actually start to see the interface come together. I'll see you in a few minutes.

LISTENING FOR USER INTERACTION

Welcome back. In this chapter, we're going to look at listening for user interaction. We're going to go through some basic event listeners. But you can find out more information about event listeners in the JavaScript Foundations course.

Returning to the plan, we have two interactions that we need to handle. One is a tap on the Menu button to open the drawer, and the other is a tap outside of the menu to close the navigation drawer. So we already have a reference to the menu opener and the menu closer that we saw when we were using getElementById a few chapters ago. Once we have a reference to those elements, we can add an event listener to them.

Now I'm going to break down what exactly addEventListener is. So addEventListener is a method available on any HTML element. And within that, we have three arguments. The first argument is the event we want to listen for. And an event is something that the user does. So it's a tap, it's a click, it's a drag, and so on. We'll look at a couple of different event types in a moment.

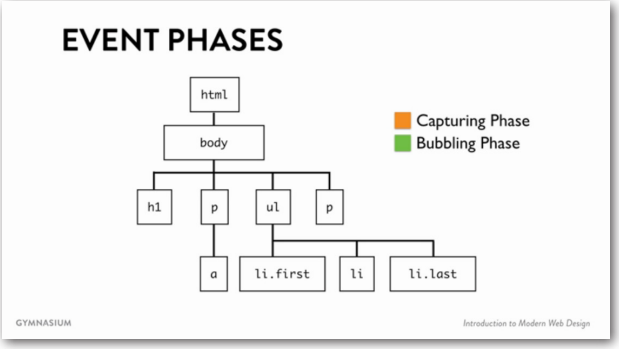
The next piece there, the second argument, is the function that we want to call when it happens. We refer to that as the event listener, hence addEventListener. And then the third argument is whether or not it should run in the capture phase.

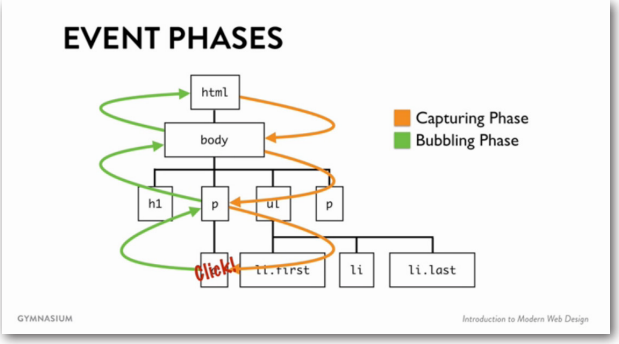
Now I'll walk through exactly what the capture phase is in a moment. But first, let's look at a couple common events.

One that should make immediate sense to you are things like keypress, keydown, keyup-- all of those have to do with the keyboard. mouseover, mouseout have to do, obviously, with a mouse moving over or off of an element. load takes place when the page loads or when an image loads. submit is often used with forms. It's triggered when a form is submitted. click is triggered when you click on an element using your mouse or a finger.

Drag and drop are events that happen when you're using a drag and drop interface. focus and blur are when an element receives focus, perhaps using the Tab key on the keyboard. And blur is when the element loses focus. And change is an event that's triggered when an element such as a form control has a value that changes like when you choose a different option from a Select dropdown or you toggle a checkbox on and off.

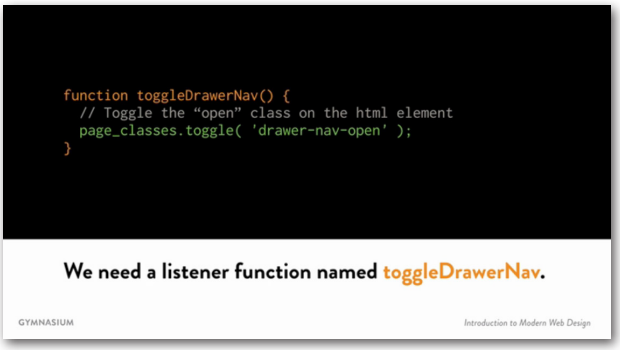
There are actually two phases in every event. Those are the capturing phase and the bubbling phase. Let's say you were to click on this anchor element inside of a paragraph which is obviously inside of a body, inside of an HTML element. When you click on that anchor, the click event happens on the HTML element, and then it moves into the body element, and then it's triggered on the paragraph, and then finally it's triggered on the anchor. All of that takes place in what's called the capturing phase. Then the event moves back up the tree in what's referred to as the bubbling phase, being triggered on each of those elements.





So returning to our example, when we say menu_opener .addEventListener ('click'-- we're now listening for a click event to take place. And then the function that we want to call when that click takes place is toggleDrawerNav. And we don't want it to take place during the capturing phase. We want it to take place during the bubbling phase. So we can use false as the third argument.

Now we've said that we're using a function called toggleDrawerNav, but we haven't created it yet. So we need to create that listener function. Now the simplest thing we could do is say page_classes.toggle('drawer-nav-open') and call it a day. The problem is that the two anchors that open and close the navigation menu have hrefs, and those hrefs bring the user to the menu or back up to the banner of the page. In other



words, they have default behaviors. So we need to find a way to prevent those default behaviors from happening. Otherwise, we're going to make our users jump all over the page.

We can do that by referencing the event. By default, when you use addEventListener, the event that takes place is passed in as the first argument. And those events have a preventDefault method which allows us to prevent that default behavior from happening. So I can simply say event.preventDefault() and it will stop the anchoring behavior to those IDed elements.

Now it's also worth noting that on mobile touchscreen devices, click is often delayed by 300 milliseconds. The reason for this is that in touch screens, a double tap could zoom the user in or out of the page. So click is delayed by 300 milliseconds in order to allow for somebody to potentially double tap.

There are touch-based events, such as touchdown, that do not have that 300 millisecond delay. So in order to accommodate both a touch screen and a mouse-based interface, we should cover both bases by assigning that same event handler to both events. So in this case, either a click or a touchdown will trigger the toggleDrawerNav method.

```
var toggleDrawerNav_running = false;
function toggleDrawerNav( event ) {
  event.preventDefault();

  // check to make sure it's not already running
  if ( toggleDrawerNav_running ) { return; }

  // block any secondary requests until we're done
  toggleDrawerNav_running = true;

  page_classes.toggle( 'drawer-nav-open' );
}
```

We can account for this in our code.

GYMNASIUM Introduction to Modern Web Desig

The problem is that touch devices do eventually fire that click listener. So they could actually fire it twice. So we need to figure out a way to throttle that to ensure that the listener function only runs once. We can do that by defining a variable toggleDrawerNav_running and setting it to false by default. And then within the toggleDrawerNav function, we can check to see if toggleDrawerNav_running is true. If it is, we can just use a return to exit that function. Or if it's not, it would continue on and set toggleDrawerNav_running to true.

Once I've done that, I still need to return it to false after the event has completed. So I could set a timeout, which I'll walk through in a moment, to turn it back to false after a certain amount of time. The setTimeout method takes two arguments-- a callbackFunction and then the number of milliseconds that you want to wait before triggering that callbackFunction.

```
setTimeout(
  function(){
    toggleDrawerNav_running = false;
  },
  500
);
```

In our case, an anonymous function resets the variable after 500ms (or .5s).

GYMNASIUM Introduction to Modern Web Desig

So in our case, I'm creating an anonymous function that just sets toggleDrawerNav_running back to false again. And I'm going to run it after 500 milliseconds or half a second. So that's beyond the 300-millisecond delay that happens in touch devices, and it also happens to correspond with the duration of the animation for the drawer to slide out that I'm using in my CSS.

So rolling it all up together, here is the whole shebang. We've got the toggleDrawerNav_running variable, which is set to false by default. That will inform our JavaScript as to whether the DrawerNav is currently toggling open or closed. Then we have our function definition for the toggleDrawerNav() function and the argument of the event.

We then prevent the default behavior of the event. We check to see if toggleDrawerNav is currently running, and if it is, we exit the function. If we know that it's not currently running, we set it to be currently running by setting the toggleDrawerNav_running variable to true. Then we go ahead and actually toggle that class on the page classes. And then finally we flip toggleDrawerNav_running to false after half a second--the end result being what I'm going to show you here in the browser.

So if I inspect, you can look at the HTML element here. Here it says the class of toggleDrawerNav enabled. If I shrink it down, now I can open and close that Menu which is pretty cool. It may look like we're pretty close to done, but we actually do still have some work to do which we'll do in the next two chapters.

What I'd like you to do now before we jump over to that is to do assignment 3, which is to play around with events on your page. Specifically, I want you to open up Chrome, open up the console as you've done before, and use addEventListener to assign a click handler to the HTML element, the body element, and a paragraph element that triggers console.log(this.nodeName). Make sure the third argument is false. Now what happens when you click your mouse on that paragraph?

Now assign a second set of the same event handlers with the third argument set to true. What happens when you click now? Post your findings in the forum. So in this chapter, we covered listening for user interactions. In the next chapter, we're going to look at how to include third party scripts into our JavaScript. I'll see you in a minute.

INCORPORATING 3RD PARTY JAVASCRIPT

Welcome back. In this chapter, we're going to look at how to include third party JavaScript into our pages. So this is what we're looking at so far in terms of the JavaScript we've written together. You see our classList test at the top. We define some variables to refer to some elements within the page and the classList API. And then we have our event listeners. And then we have the definition of our toggleDrawerNav() function.

Now, returning to our plan, we know that this stuff is only supposed to run if the screen is small. So how do we do that? I don't want to write this from scratch. So I can take a script that I've actually written previously and post it as a gist on GitHub. You can see the URL here on your screen.

Now, what I've done is I've created a global method called watchResize() that we can use to watch for resize events and call a particular function. So I can incorporate that into my JavaScript directly, because it's pretty small. I just drop that code into the existing JavaScript that I've made and that makes the watchResize() function available to me. If this was a particularly large JavaScript, I might choose to keep it as a separate file and create a new script element.

Now, in order to ensure that I can use that script immediately when this script runs, I would need to include that script element prior to my main.js file. So for simplicity's sake, I'm just going to go ahead and directly put that code at the top of my document here. So window.watchResize is equal to a function. And you can get the complete code for that at the gist that you see there in the comment.

```
window.watchResize(function(){

  /* the contents here will run any time the browser
     is resized (and also on load) */

});
```

Using watchResize() is pretty straightforward. It's basically just a wrapper for an onresize event listener on window.

GYMNASIUM Introduction to Modern Web Design

So using watchResize() is pretty straightforward. It's basically a wrapper for the onresize event for the window. But it also includes some throttling. And it also runs once when the page finishes loading.

So in our script, we could simply say window.watchResize, and then inside of that pass a function as the argument. And then the contents of that function would be executed every time the browser is resized. So you can try this out yourself by pasting that gist into the console and then typing this code into the console-- window.watchResize, and then inside of the function, console.log('hi'). Then when you resize your browser, it will keep saying "hi" over and over and over again.

```
window.watchResize(function(){

  // if the screen is 'small'

    // handle the click & touch events
    // add the "enabled" class

  // otherwise

    // ignore those events
    // remove the "enabled" class

});
```

Here's the basic logic outline.

We're going to use the watchResize() method to help us figure out if the screen is small. We'll do that by setting up some comments here for what it is that we want to do. If the screen is small, we're going to take one path. And we're going to handle the click and touch events and add the "enabled" class. Otherwise, we're going to ignore those events and remove the "enabled" class. This is the basic logic outline for the program.

```
window.watchResize(function(){

  if ( window.innerWidth <= 648 ) {

    // handle the click & touch events
    // add the "enabled" class

  } else {

    // ignore those events
    // remove the "enabled" class

  }

});
```

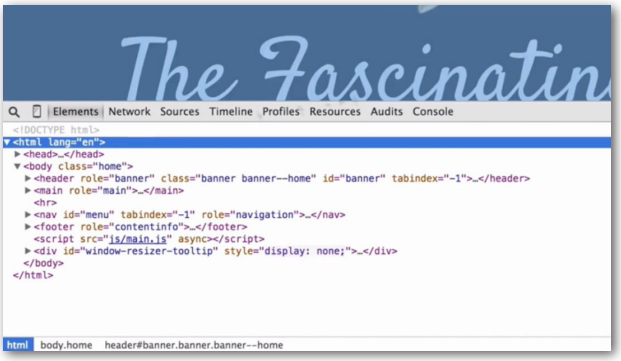
First we can fill in the conditional. Then we can add what happens in each case.

GYMNASIUM Introduction to Modern Web Design

First things first, let's fill in our conditional. In my CSS, my small breakpoint ends at 40.5 em's, which is equivalent to 648 pixels. So I can use that value here. I can say if window.innerWidth is less than or equal to-- that's another comparison operator-- 648, then take the first path. Otherwise, else, do the second path.

Then I can go ahead and move my event listeners that we created in the last chapter into that conditional. Now, we have addEventListener(). The corollary to that to remove an event listener is removeEventListener().

So I can do exactly the same thing with removeEventListener() in the else branch of this conditional. That allows us to unassign those event handlers and set everything back to the way that it was. And then, of course, I need to add and remove the class name drawer-nav-enabled in each of those scenarios.



And all told, we end up with something that looks a little bit like this. Hopping over to the browser, we can take a look, inspect the element. And now you see that the HTML does not have that class name assigned to it. But if I shrink the browser down, once I get below 648, bam-- there's the class added. If I increase it again, the class disappears. Pretty cool.

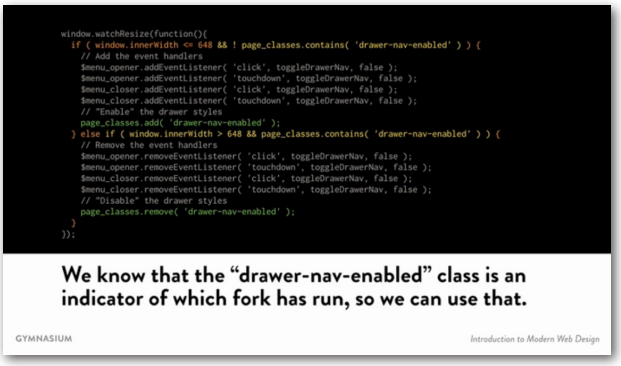
All right. So that was a pretty brief chapter, but it demonstrated how we can include and use third-party scripts. In the last chapter of this lesson, we're going to look at refactoring our JavaScript, because right now, it's not actually very efficient. So we're going to make things a little bit better. I'll see you in a few minutes.

REFACTORING JAVASCRIPT

All right. Welcome back to our final chapter in Getting Started With JavaScript. Now, we're going to be focusing on refactoring our JavaScript. In other words, we're going to go through and we're going to edit it to make it better. We're going to do this by identifying problems in the code and then fixing them.

So the first problem that we're having in the code is that we're firing events when we don't need to. If you remember, in our resize watcher, we're checking to see whether the width is less than or equal to 648 pixels or if it's more than it. Now, that's not terribly efficient, because it will happen every time the browser resizes. So if the browser is resizing from 648 to 646, it will run that first branch again, even though it's already run. So that's not terribly efficient.

We do, however, have the drawer-nav-enabled class that can be an indicator of which fork has run. So we can use that and incorporate that into our conditionals. So instead of just saying if window inner width is less than 648, I could say, if window inner width is less than or equal to 648 and-- that's what the double ampersands means-- and page classes does not contain drawer-nav-enabled, follow this path.

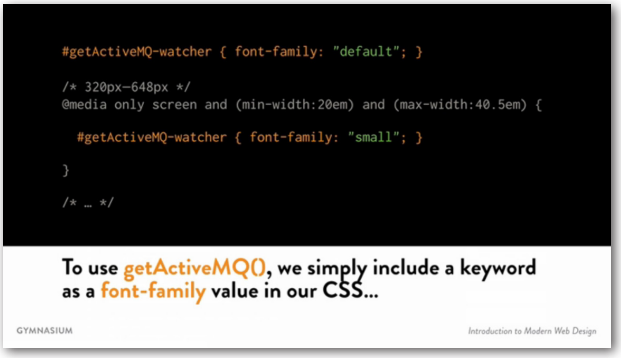


And then I'm switching the else to an else if statement, so I can supply additional conditions for that branch. So otherwise, if the window inner width is greater than 648 and the page classes does contain drawer-nav-enabled, then go ahead and run that second branch. So now, those two branches will only run when the browser is going less than 648 pixels the first

time or more than 648 pixels the first time. And they won't run again unless the other side has run first. So in other words, we won't be continually adding or trying to remove event listeners or classes to the page. So that's far more efficient.

Problem number two-- we're repeating ourselves quite a bit. If we look at the code that was just up on the screen, you see I've got drawer-nav-enabled four times in there. I can extract that string to a variable, as we saw in the earlier variable examples, and say drawer enabled classes is drawer-nav-enabled. Drawer open class is door-nav-open. And simply replace those strings with that variable reference. Then once defined, I can go ahead and use those variable references anywhere. And what's nice about this approach as well is that it means that if I want to change what those classes are, I can do it in one spot, as opposed to having to hunt for it throughout the entire script.

All right problem number three-- our CSS and our JavaScript are currently very tightly coupled. So in the conditional, window.innerwidth is less than or equal to 648, but that maps to the current CSS media query, not a future CSS media query. So if that media query were to shift to being approximately 660 pixels, then the JavaScript and the CSS would get out of sync. That's a problem.



In order to address this issue, I created another bit of JavaScript called getActiveMQ.js. JS. In other words, get the active media query. And the way that this particular method works is that you add some styles to your CSS for something with an idea of getActiveMQ-watcher. And in that CSS, you declare a font family that is a text string value. So in this case, font-family default. And then within my small media query, font-family=small.

And then if I embed that JavaScript into my file, I can go ahead and use that method to find out what the current media query is. So I can save var current_MQ=window.getActiveMQ. And then I can test if current_MQ is equal to small and the page classes does not currently contain that drawer enabled class, then I follow branch number one. Otherwise, if current MQ is not equal to small and it does contain drawer enabled class, then do the second branch.

So that's a pretty nice refactor that makes it much easier for us to evolve our CSS without worrying about our CSS and JavaScript getting out of sync. That's pretty much it for what I wanted to cover in this chapter. But before I give you your final assignment, I wanted to give you some next steps to help advance your learning and possibly your career.

The next logical step on Gymnasium is to explore the Responsive Web Design course or the JavaScript Foundations course. Both of these are full courses and will take you on a long, if not longer, journey than this one did. And both offer certificates if you pass the final exam.

If you're looking for something shorter and more specific, there are also Gym Shorts on subjects such as WordPress, Bootstrap, and other related topics. There are also a number of resources out there for you to explore. A couple that I would suggest is A List Apart, which is a great online magazine. Smashing Magazine is another fantastic online publication. Both have awesome tutorials and excellent information that will help you become a better web designer. I also highly recommend checking out Jeffrey Zeldman's Designing With Web Standards. It's a fantastic book. And it's an invaluable resource.

Now, on to the final assignments. First of all, I want you to take quiz number five. I want you to post your project to GitHub. At this point, you should have a complete site that includes the HTML and CSS concepts you've learned, as well as the navigation menu that we just created in JavaScript. Once you're done, post a link to the site in the forum so your classmates and the Gymnasium teaching assistants can give you feedback.

Then, I want you to take the final exam. There's a final exam ready for you to take on the classroom site. I suggest you study up on all the quizzes from each lesson in order to prepare for it. If you did well on the quizzes, my guess is you'll also do well on the final exam. And if you pass it, you'll receive a certificate that you can share with anyone who might be interested, perhaps a future employer.

Finally, thank you so much for your attention during this class. I hope you've enjoyed it. It's been my pleasure to teach. And I look forward to seeing what you do with your new skills. This has been an introduction to modern web design. Thank you.