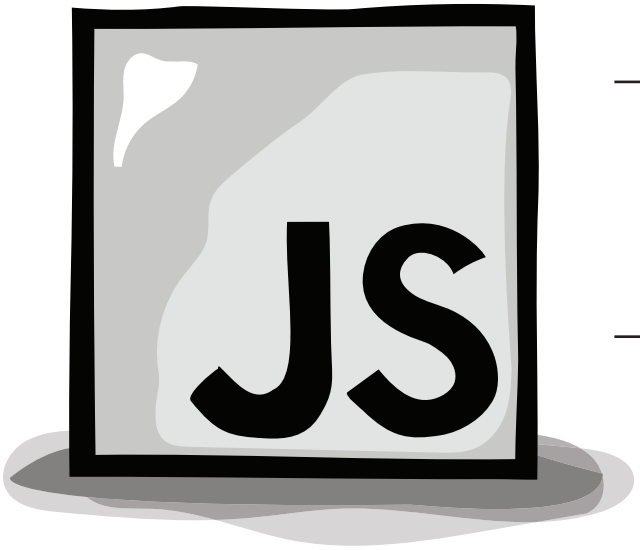



GYMNASIUM

# JAVASCRIPT FOUNDATIONS

*Lesson 3 Handout*

*The DOM (Document Object Model)*

# 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

1. HTML elements on a page are represented by JavaScript element objects.
2. Selecting an element doesn't do anything to that element. It merely finds a matching element and gives you a JavaScript element object that represents that element.
3. Element objects have various properties that allow you to inspect the attributes of the represented HTML element.
4. Changing the values of those properties instantly updates that HTML element and re-renders the page.
5. The textContent property allows you to set the text that appears inside of an element. This will replace any previous text or child elements though, so be careful.
6. The innerHTML property is similar to textContent but allows you to pass a string of HTML. This will create new child element for the selected element.
7. The style property is an object. It has many properties of its own, which represent the various CSS styles that can be applied to that element.
8. An application generally gets input, often by reading the values of existing HTML elements, validates that input, processes it somehow and outputs the result, often by setting the values of other HTML elements.

# ASSIGNMENTS

1. Quiz
2. Go to two or three pages of your choice, preferably ones with a good amount of complex text. Open the console and do several queries for common elements, such as h1, p, div, etc. Try reading and setting various properties of the elements you get, such as text content, innerHTML, and any styles you want. Remember that some selector functions will give you back an array of items, so you'll have to choose which element to work with by specifying an index. Repeat this until you are comfortable selecting different types of elements and modifying them.

3. In the sample application we created, we are validating each numeric property and setting the border style of the input related with it like so:

```
if(isNaN(miles)) {
    distanceInput.style.borderColor = "red";
    return;
}
else {
    distanceInput.style.borderColor = "initial";
}
```

Create a new function in that project called validateInput. It should take a number value and a text input element as parameters, like so:

```
function validateInput(value, input) {
}
```

Fill in the body of that function so that it performs the same action as the original code. Replace each original if/else statement with a call to that new function like so:

```
validateInput(miles, distanceInput);
```

See how this makes the overall code smaller and easier to read.

4. For an extra challenge, have the validateInput function return true or false based on whether or not the input is valid. Use that in the original code to jump out of the click event handler function if the validation fails. This might look something like this:

```
if(validatedInput(miles, distanceInput) == false) {
    return;
}
```

But there is an even more concise way to do it using the "not" operator, which is the exclamation point "!".

# RESOURCES

- Again, more documentation. The objects that represent HTML elements on a page have a type of "HTMLELement". But they are also a type of more general object just called "Element". So they have properties and methods from both of these and you might need to look in two places to see all of properties. This is the page that lists all the properties and methods of element objects:

  https://developer.mozilla.org/en-US/docs/Web/API/element

And this lists the properties of HTMLElement objects:

https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement

- Read through these and find some other interesting things you can do with elements. As you do this, think of some other ways you could use these properties or methods an in application.

  And here is the reference for the style property:

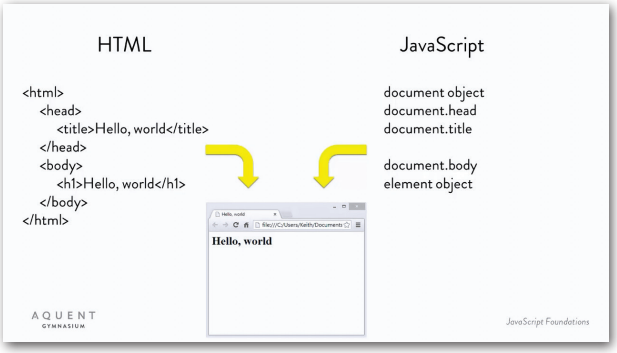
  https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement.style

# INTRODUCTION

*(Note: This is an edited transcript of the Javascript Foundations 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.)*

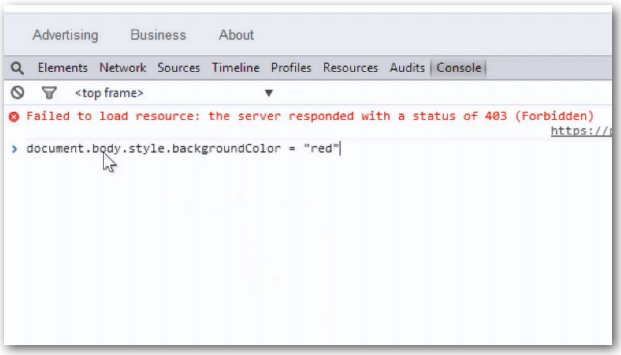
Well, I'm really glad you made it here, to Lesson Three. We covered a lot in lessons one and two; all stuff that you'll be expected to know, now that we're advancing into more interesting, and complex stuff. In this lesson, we're going to cover the manipulation of the DOM. The DOM, as you probably know, is short for document object model. This means that it's a model of the HTML document, where each HTML element is represented by a JavaScript object.

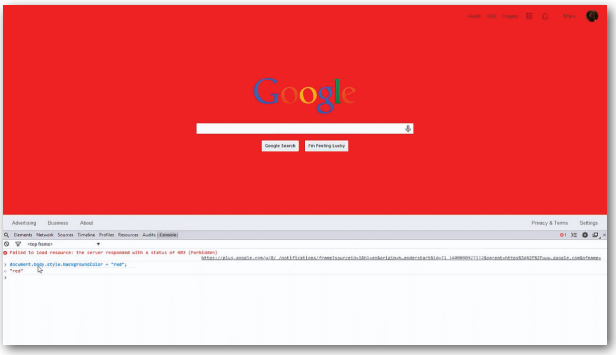

It's probably good to make the distinction between these objects, and the actual elements themselves. The elements are defined with HTML tags in the HTML document. These get loaded in, and rendered in the browser. The JavaScript engine in the browser then looks at this page, and creates a series of JavaScript objects that represent those elements. The objects are not the elements. But, the objects let you access, and manipulate those elements. It's kind of like, a friend icon on a chat application. The icon is not actually your friend. But, if you click on the icon, it lets you communicate to her.




In JavaScript, any changes you make to an HTML element object are immediately reflected on the HTML page, itself. To see this in action, open up a browser to a simple page, like Google.com. Then, open the JavaScript console. We'll be getting a lot more into changing styles with JavaScript, a bit later in this lesson. But, I'll give you a quick example of changing the background color of an element.

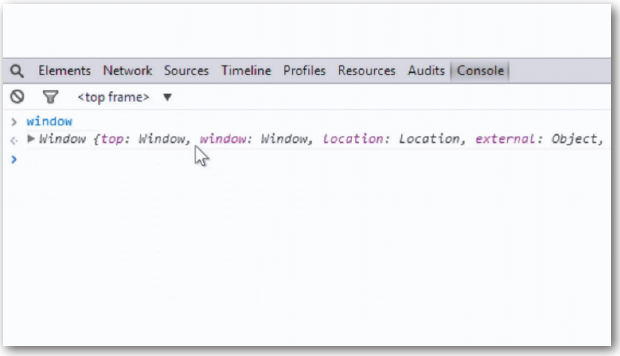
So, type document.body.style.backgroundColor = "red", and the background turns red. All element objects have a style property that contains all the CSS styles for that element. Each style is a property on that object that you can use or assign values to. Again, we'll be doing a lot more on that, later.



For demonstration purposes, I've created a simple HTML document that we'll use throughout this lesson. It's called dom.html, and you can download it from this lesson's page. And, open it up in Chrome, so you can follow along here.

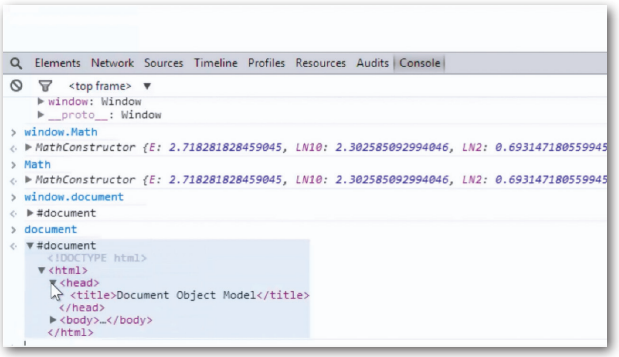
Now, there are a couple of things you should know about the DOM from a JavaScript viewpoint. First of all, the top level object for any HTML page is the window object. This represents the window of the browser that the page is loaded into. Open up your console, and type the word "window." You'll see that this is a window object, and it's even giving you the object syntax showing you some of the properties with the property name, colon, and value. But, if you click this little arrow here, to the left, it expands, and you can see all of the properties of the window. As you can see, there are a ton of them.



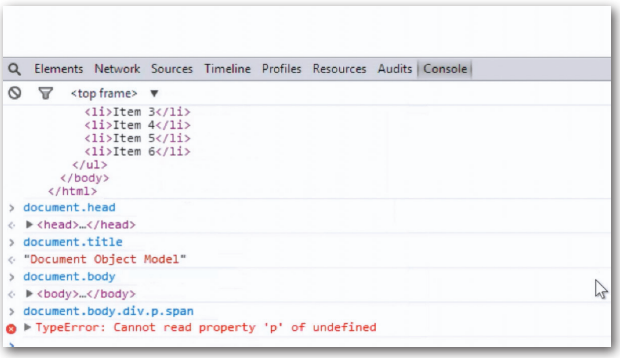

Basically anything, and everything, JavaScript needs to know about when it's running in a browser is available from this window object. Here, you can even see the math object that we've used a few times. Expand that, and you can see all the properties, and methods of math. So, math itself is our property of the window object. You can type "window.Math", and there it is. But, you can also just type "math" and get the same thing.

The reason that you can do this, is that there's so much stuff on the window object, that rather than forcing you to type "window" every time you need access to it, the window part is just assumed.

Another important property of window is document. You can type "window.document", or just type "document." We used this a little bit in lesson two to listen for mouse, and keyboard events. Again, document represents the HTML document itself. So, if you expand this, you can see the document exactly as you just saw it in the editor. You can expand the individual parts of the document, and see what's in them. Now, there are certain elements that just about any HTML document has; namely, a head with a title and body. These are available directly as properties of the document object. So, you can type document.head, and see what's in there. Or you can type document.title, and see the title we've given this. And, then there's document.body.

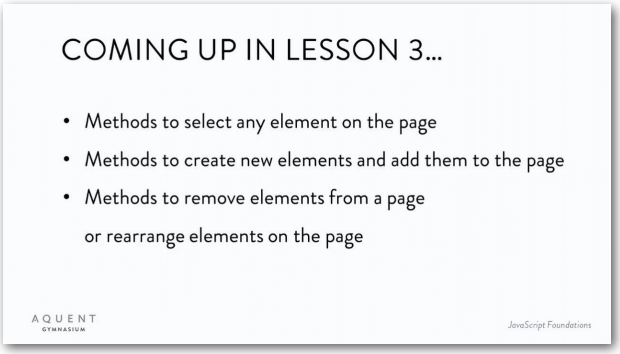



But, beyond that, all web pages start to vary, and you can't really assume anything. So, to get access to everything else in the page, you can't use built-in properties. You can't just say something like, document.body.div.p.span. Your particular document might have a div with the paragraph containing a span, but not every document will.



But, in our document here, there are certainly lots of other elements that we'd probably like to do something with. So, how do we get access to those? Well, there are various JavaScript functions that will search through the document, and find the element, or elements, you're looking for. Those methods generally return either a single element, or a list of elements.

Either way, each element is a JavaScript object that represents an actual element on that page. We'll get to those functions a bit later in this lesson.

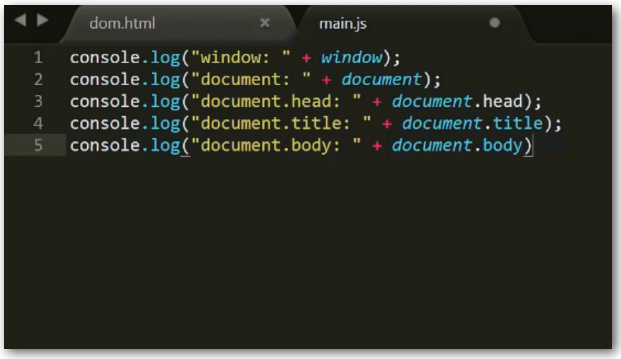


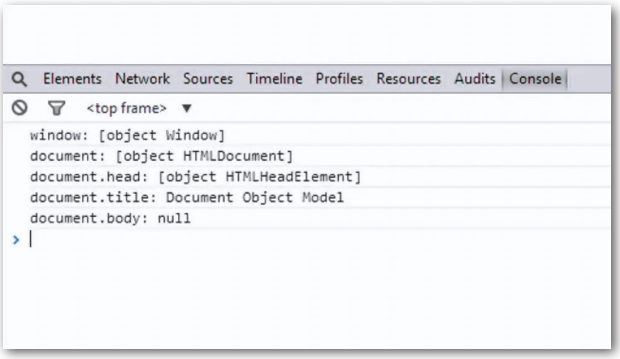

In addition to simply finding existing elements, JavaScript will also allow you to create brand new elements, and add them to the page, and remove existing elements for the page, or even rearrange the structure of elements on a page. This gives you all kinds of power to control what is on the page with code, alone. We'll cover a lot of that in the next lesson. So, there are some really good things in store, here. But first, we need to take a look at what happens when a web page loads, and when is it ready for us to examine, and manipulate with JavaScript?

## DOCUMENT READY SOLUTIONS

Before we start running code that tries to access HTML elements on a page, we need to make sure that the HTML page is actually loaded in, and those elements are available to us. We obviously can't access elements that are still working their way across the internet, into our computer. So, in this video, we'll take a look at what happens when an HTML page loads. I've called this section, Document Ready Solutions, based on the jQuery function used for this purpose. It lets you know when the document is ready.

To start, take another look at the DOM HTML file, here. In the head there's a script tag loading a JavaScript file called, "main.js." That's an empty file, right now. Go ahead, and open it up in your editor, and let's log some of the elements that we just talked about. We'll do a console log window, and document, document head, title, and body. Now, let's run this in the browser, and check the console. Okay, everything looks fine, up until document body. That's undefined, for some reason. Well, there's definitely a body tag in this page, so what's going on?

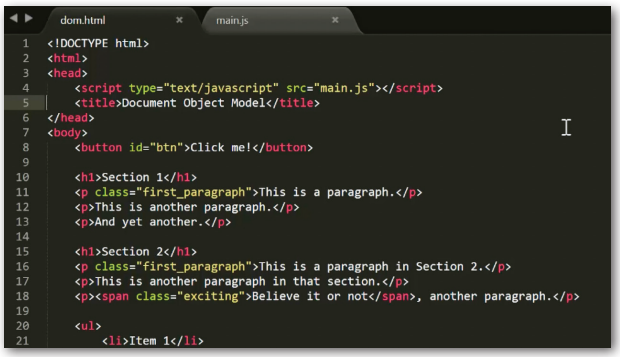


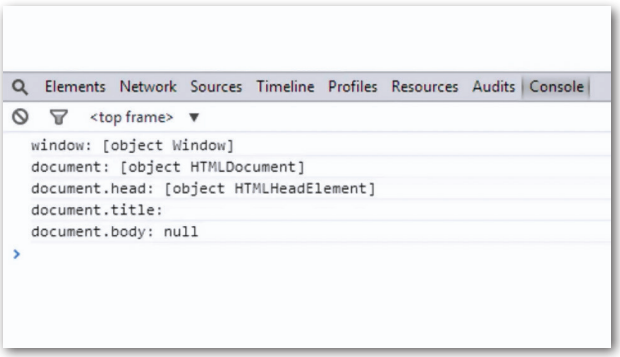




Well, the key is right in the structure of the document. Look where that script tag is. Everything before that, the document, head, and title, is defined before the script. But, the body tag comes after the script. It turns out that the browser doesn't load in the entire HTML document, render it all, and then, run any JavaScript on the page.

Instead, what it does is, it processes the document exactly as it's written. When it sees the head tag, it creates a head element in the browser. When it reads the title, it sets the title on the page. When it sees a script tag, it stops everything, loads in that JavaScript file, and executes it. When that JavaScript is done executing, only then does it continue on, processing the rest of the document. It sees the body element, and creates that, and then works its way through the rest of the tags. So, yeah, when that script runs, the body really just doesn't exist, yet.

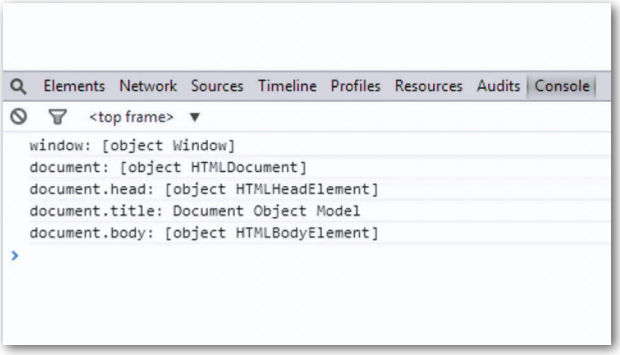
Let's try moving this script tag up here, before the title, and run that again. Now, only window, document, and head are defined. The script is running before the title is created. Although title is not undefined in this case, it's still an empty string, not the title we said in the HTML. So, what do we do? Our code may need to access the elements within the body, but when that code runs, none of those elements have even been created, yet. Well, there are several solutions, and I'll go through them, and give you the pros, and cons of each.



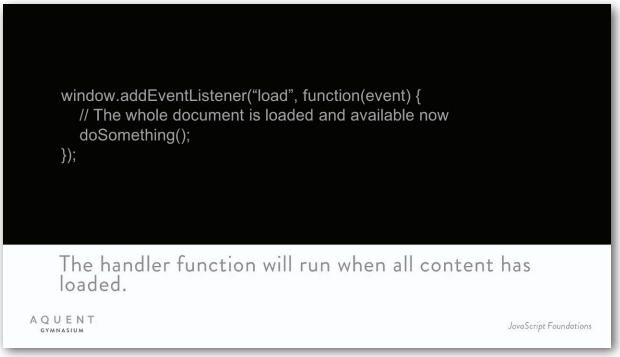

First of all, if you recall back to lesson one, where I was showing you where to put your code, I created a script tag in the body element after the button, so that the script would have access to that button. This may seem odd, if you're used to seeing scripts in the head of an HTML document, but there is no requirement that says that's where they need to live. So, let's move this script down to the bottom of the body, and run it again.

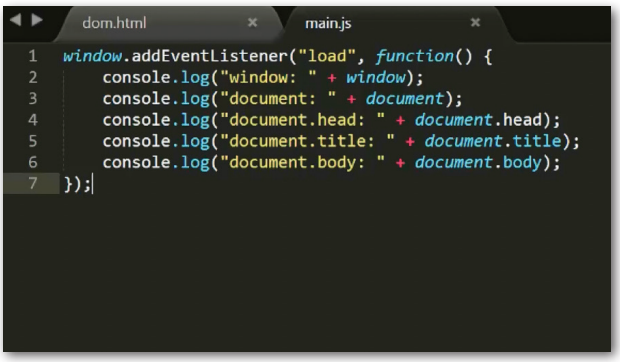
Well, that's better. Our document now has a body. While this may seem too simple of a solution, it's actually a recommended practice by many developers. When you put the script tag in the end of the body, the HTML loads, and gets rendered, and then the script runs. So when that script does run, you'll know you'll be able to access all of the elements that are on the page.




The second solution is using the window load event. In lesson two, I talked a lot about mouse and keyboard events, but I mentioned that there were other events you'd be learning about as well. Well, here is an event called "simply load." This occurs when an HTML page is fully loaded. You can listen for this right on the window object, like so.

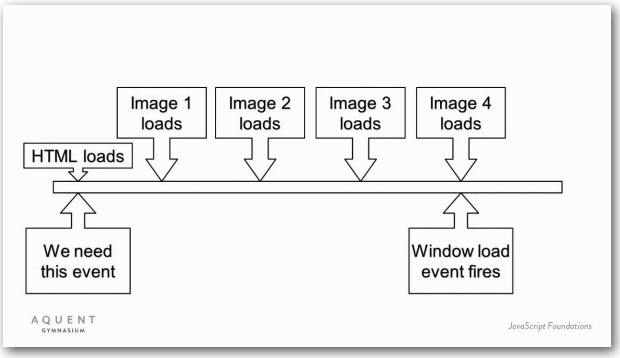
So, let's go back into the code, and move that script tag back up into the head. Then, we'll wrap those log statements inside of a load event handler. Now, this main js file will load, and execute before the rest of the document is created. But, all it does is create this event listener. It's not trying to access anything in the DOM, other than window. So, when the rest of the document loads, the load event will trigger. The function will run, and then that code will attempt to log out all those DOM elements.



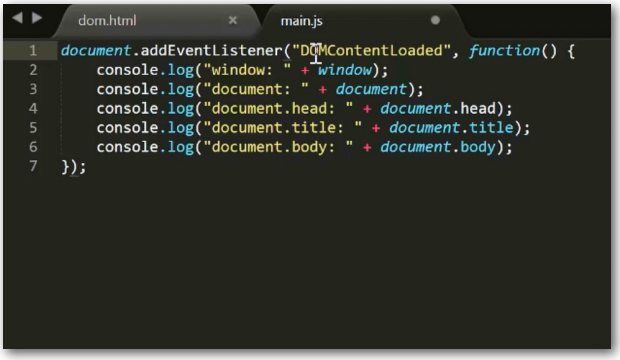

And, here you can see that everything is defined. Well, great. That seems pretty cool. Why not just go with that one, and be done with it? Well, there is one drawback with the load event. It doesn't just trigger when the HTML document, itself, is loaded, and rendered. It waits until everything that's referenced on that page is also loaded. That means that all the assets, images, CSS, anything else on that page has to load first, and then the load event triggers.

So, if you have a graphically heavy page with large images, and a slow connection, your code may not run for several seconds, while all those images work their way down to your computer. This may, or may not be, a problem for you. It all depends on the site. For a largely text-based site, this might work fine. For a site with lots of images, and other assets, maybe not so good.




What you really want is an event that says the HTML has loaded, and has been parsed. Maybe all the images and other assets that are associated with the page are still on their way, but the document itself, is ready. Well, there's another event for that called, DomContentLoaded. As I just said, this triggers when the HTML document itself has been completely loaded into the browser, and is ready to go.

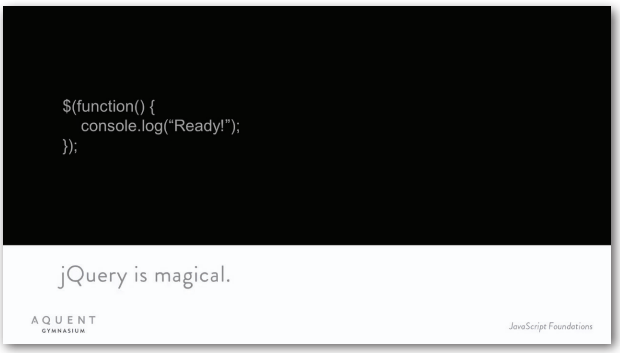
From a programming viewpoint, that works pretty much the same as in the load example. Just change the element to document, and the load event to DomContentLoaded. Run it, and we're good. Well, hey, that one seems pretty cool, too. Right? Even better than the load event, so let's just use that all the time. Yeah. So, about DomContentLoaded. You can use it, and for the most part, it's going to work just fine. But, it might not work on every browser. It should work on all modern desktop browsers, including Internet Explorer nine, or later. But, it's not supported in IE8, or before. And, for mobile browsers, about the same. It should work in most of them, but possibly not all.



Problem is, you don't want your site to mostly work, especially if you're basing all of your JavaScript on this one event firing. So, this one is a bit risky, and I can't 100 percent recommend using it at this point. But, there's one more solution, jQuery. I'm not going to take a deep dive into jQuery, itself, here, but it does have a built in way of adding a listener for the DomContentLoaded event. And, one of the cool things about jQuery is that, if a feature is not implemented in a particular browser, it'll resort to other workarounds, to make sure that you get the same behavior in every browser.

In jQuery, for example, you'd write something like this. The jQuery syntax can seem a bit confusing at first. That dollar sign is a single letter variable representing the jQuery library, which itself is just a very complex function, with all kinds of other functions, and objects associated with it. Here we're passing the document element to jQuery. This returns an object that has a ready method. And, you pass a function to that ready method, and that function gets called, when the DOM content is loaded.


Internally, jQuery does use the DomContentLoaded event in browsers where it's supported, but uses alternate methods in other browsers. So this should always work. In fact, through some other behind-the-scenes sleight of hand, you can actually do the same thing in jQuery with this code. I'm not even going to try to explain how that works. Like so much of jQuery, you just have to trust that it does.




So, I've just given you a bunch of different strategies. Which one should you use? Well, there's no right answer. If you're already using jQuery in your project, you might as well just use its ready function. But, I wouldn't add jQuery to a project if you're only going to use it for its document ready functionality.

If you have a lightweight page, without a lot of images or other large assets, you can probably get by with listening for the load event. Just test it to make sure your page isn't sitting in an unresponsive state for a long time, while stuff is loading.

If you can't decide, you can't really go wrong by putting your scripts at the bottom of the document. This is simple and foolproof, and has essentially the same behavior as DomContentLoaded. This is the technique that I'll be using for the rest of the examples in this course.

But no matter which one you wind up using in any particular project, it's important to be familiar with all of these methods, as you'll likely run across all of these in code that you work with.

So, now that we have a document loaded and accessible from JavaScript, let's get out those elements.

## SELECTING ELEMENTS

All right, now we have our code running. And, when it runs, we know that the DOM is locked, and loaded, and ready for action. We can get at the window, the document title, head, and body easily enough. But, how do we find the rest of the elements on the page?

Well, first of all, let's deal with the obvious answer, use jQuery. Selecting elements on a page, and then manipulating them in some way, is a huge part of what jQuery is all about. It's certainly very powerful in that regard, but this course is not about jQuery.

If anything, I'd like to teach you some alternatives to jQuery, so that you're not totally dependent on it. Actually, some of the methods we'll look at here are exactly what jQuery uses under the hood. Like with document ready, it uses certain methods on browsers that support them when those methods are not supported.

So, what we want to do is select elements on the page. Let me clarify here, that when I say, select an element, I don't mean it in the sense of selecting a checkbox, or radio button. We're not doing anything to the element when we select it. We're not highlighting it, or putting it into some kind of selected state. By selecting, I really just mean, getting the JavaScript object that represents that element.

Let's start with one of the simplest selection methods, get element by ID. This is a method on the Document Object. If you have an element on the page with an ID, you can say, document.getElementById, passing in a string that corresponds to the ID attribute of that element. Now, I say that this is one of the simplest methods, because it will return, at most, a single element. Of course, if you pass it in an ID that doesn't exist on the page, it will return undefined.

```
<button id="submit_button">Submit</button>
```

An element with an id.

But, barring that, it will always return a single element. This is because, technically, every ID on a page should be unique. That doesn't mean that every element needs to have an ID, but no two elements on the same page should have the same ID. However, even if you goofed, and did assign the same ID to two elements, getElementById would only return the first element on the page with that ID. Let's try it out.

```
<button id="submit_button">Submit</button>

var btn = document.getElementById("submit_button");
```

getElementById returns a single element.

On the page here, you can see there's a button with the ID btn. So, we can go back into the main JS file, and remove all those trace statements. And, instead say, var btn equals document.getElementById("btn"). Now, we have an element object that refers to that button. We can do whatever we want to it.

The next video will cover accessing, and changing element styles and properties. But, you already know how to do at least one thing with elements, addEventListener to them. So, let's add a Click Event Listener to this button. And, in the handler function, we'll just throw up an alert box with a message.

And, yeah, so this is exactly what we did back in lesson one. But, now, you should understand every single line, and every word in this program. You've come a long way.

The next method we'll look at is getElements by class name. So, you have one or more elements on the page that have a certain CSS class applied to them. You call getElements by class name, passing in a string with that class name, and it will return all the elements that have that class.

So, notice here that it's elements **plural**, not elements singular. There's no limitation on how many elements can have the same class in a document. So, when you search for elements by class name, you might get any number of results.


So, the return value for this is an array. Well, technically, it's called an array-like object. You use it just like an array, with square brackets, and indexes, and the length property, but it doesn't have all the properties and methods that a full array has. But, for most of what you do with this object, you can consider it an array.

Now, here in the document we have several paragraph elements. A couple of them have the style first paragraph, so, let's select those. Say, elems equal document.getElem entsByClassName( "first_paragraph").

Again, we'll get into styles more later, but we can quickly change the background color of those paragraphs by setting the Style Background Color property. We'll have to loop through the array, and get a reference to each paragraph object. You should already be familiar with using for loops to loop through arrays from lesson two. Now, each element in that list will be referenced as elems index i. Once we have that, we can set the background color.



And, here's how that looks. Each first paragraph has a red background, so yeah, we selected the right ones. Another similar method is getElementsByTagName. You should be able to guess how this works. You just pass in a string with the tag name that you want selected.

This HTML has a list with several items in it. Let's grab those list items by passing in li. Now, here's a neat trick; rather than incrementing the for loop by one, use plus equals two. This will give us a reference to every other element. And, styling the background colors for those gives you alternately colored rows. Could be a useful effect.



The next two methods I want to cover are called, querySelector, and querySelectorAll. These both allow you to specify a query that will select certain elements. But, querySelector will select the first element that matches that query; whereas querySelectorAll will return an array-like object, containing any and all matches.

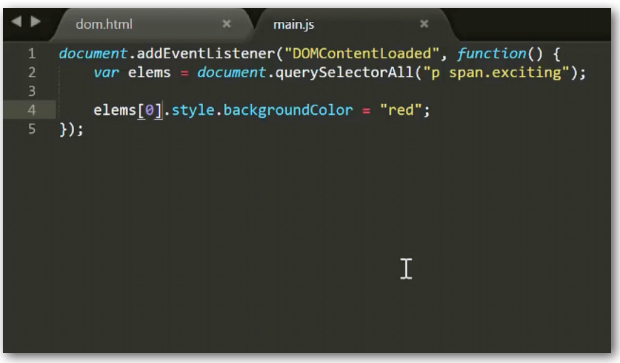
So, what is this query? Well, if you're familiar with CSS, you should be right at home here, because these queries are simply CSS selectors. So, you can pass in a tag name like ("p"), paragraph. Or a CSS name with a dot, like .first_paragraph. Or an ID with a hash, like ("#submit_button").


Or, you can use more complex, combined CSS Selectors. Here, we have a paragraph with a style of article inside a div. If you use jQuery, you'll also be right at home here, as this is mostly what jQuery relies on for selecting elements. You should know that, these methods are not supported in Internet Explorer seven or earlier, and IE eight only supports CSS2 Selectors with these methods.

So, let's use this to create a bit of a more complex selection. In this paragraph here, there's a span with a class of "exciting." So, we can use a querySelector with a selector of ("p span.exciting"). This query will give us that element, and we can style this background color to red.

Now, we could also have used querySelectorAll here. But, it happens that there's only a single element in the whole documentary that will match the Selector. And, I said that querySelectorAll will return an array-like object of all the results.

So, which is it in this case, one element or a list? Well, it's both. It's an array containing that single element at index zero. So, here, we could either loop through it, if we think there might be more than one, or we could just access it with index zero. And, you see that gives the same result.




So, there you go, several different methods for selecting elements. Which one should you use? Well, like the (document).ready solutions, there's no single, right answer. If you just need to select a single element, and that element has an ID, then getElementById is perfectly fine for that task.

And, of course, if you want to select all elements of a certain type, or those with a certain class, then getElementsByTagName, or getElementsByClassName will work nicely. Of course, you can do the same thing for all of those with querySelector, or querySelectorAll, as well. But, if you need to do it in a more complex selection, then querySelector methods are what you really want. Just remember that, if you're targeting anything before Internet Explorer eight, you might be in a bit of trouble.

And, of course, if your project already has jQuery in it, you can just go ahead, and use that. But, hopefully you're seeing that there's a whole lot you can do already, without depending on jQuery. So, now that you know how to select elements, and get the JavaScript element objects associated with them, we'll next look at what we can do with those element objects; in particular, reading, and setting various properties, and styles.


## ELEMENT PROPERTIES AND STYLES

Now that we're able to select any element on a web page, let's take a look at what we can do with those elements. Remember that the selection methods give us back JavaScript objects that represent HTML elements. And objects can have properties, which holds values, and methods, which are functions that can be called to perform actions. So, element objects have several useful properties.


Pretty much anything you can do to an HTML tag, you can do with JavaScript, via an element object. For example, you could read the elements ID, or class, or tag name with the ID, class name, and tag name properties. You can even change some of those properties, like ID and class name, so you can swap in and out CSS styles of an element very easily.

A couple of other very useful properties are text content, and inner HTML. These allow you to read, or change the content of an element. Let's open up the same sample HTML file we've been using for this lesson. Open up the console, and let's select the first list item on the page, using querySelector. Now that we've selected that first list item, let's change its text content. I'll change it to the string, "thing one."


Notice that the list item itself is instantly updated to reflect the text you passed in. Be careful what you use this on, though. If the element you've selected has any child elements, this will wipe them out. For example, if we select the whole list, and change its text content, now the list is gone, replaced by the text. If we open up the elements tab, and look at that, we see that yes, indeed, that list now only contains the text we set. Those list items are no more.

Now, inner HTML performs a similar function, but allows you to add new HTML tags, as child tags of the selected element. So, we could create some new list items in here, like so. And, instantly, our list has items. If we look back in the elements tab, you see that this new HTML has replaced the previous, inner text. So, this is one way of actually creating brand new elements, and adding them to a live site in real time.

In the next video, we'll look at more powerful methods of creating, and adding new elements. But, I hope you're starting to get an idea of how powerful JavaScript can be for creating a dynamic site. Next, let's look at one other property, style. This is right up there with text content and inner HTML, as one of the most powerful HTML element properties, as it allows you to change, add, and remove CSS styles on an element to add run time.


This allows for very dynamic pages, and applications that change the look and feel, based on the actions the user is performing. Some examples you might be familiar with are showing or hiding elements, or adding a red border around an input field to indicate that the user has not filled it in correctly. The style property, itself, is a JavaScript object, and it has many properties of its own. These properties all correspond to the possible CSS styles that can be applied to an element.

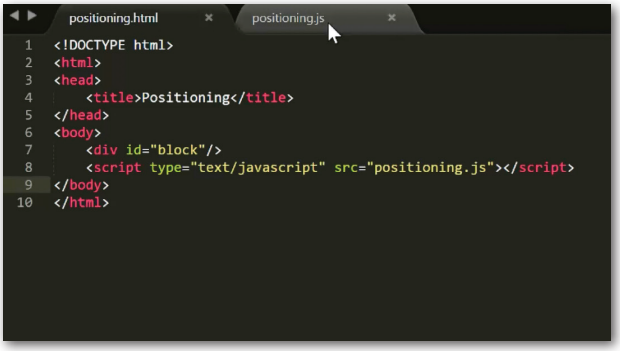
This page here, lists many of the styles. You can see here that some of the property names are a bit different than the style names. In particular, many CSS styles contain hyphens, such as background, hyphen, color. This would not be a legal JavaScript property name, so it's redefined with camel case, backgroundColor.

Now, go back to the browser, and reload that page to clear out any of the changes we made earlier. Let's select the first H1 tag, with querySelector, H1. Now, we can change that font family to Ariel, and the font size to 48 pixels. Notice that, you have to specify the units in pixels, points, ems, et cetera, just like you would in CSS. And, we can change the color to red.

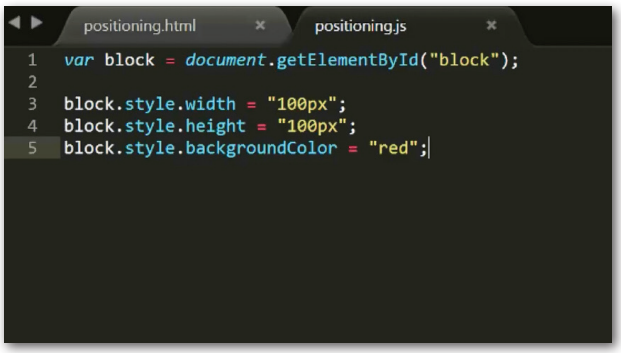

We can also set the color, using RGB values, like this. This looks, and works, like a function, but notice that it's actually a string. You insert the values for red, green, and blue with integer values from zero to 255. This string changes the text color to green. Note that, each time we change a style or property, the HTML document is updated, and rerendered instantly.

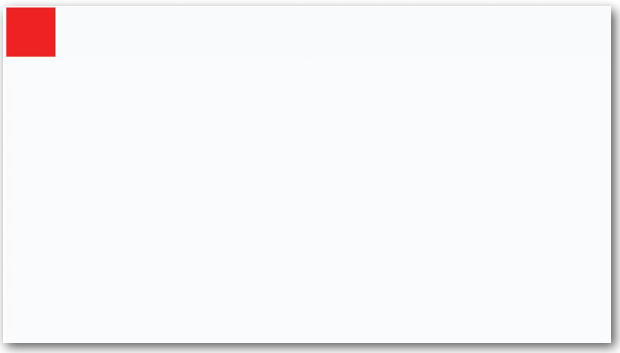
This will become very important in lesson four, when we're dynamically creating many elements. Rerendering every time you change a style, or add an element can slow things down, so sometimes we want to hold off on that rendering until we're completely done. But again, that's a preview of lesson four.

Another interesting, and useful thing you can do with the style property is change the positions of elements. Here, I've set up another file called, positioning.html. This just has a single div in it with an ID of "block." It also loads a positioning.js file. Take a second to download these from the course's page, and open them up in an editor, and browser.



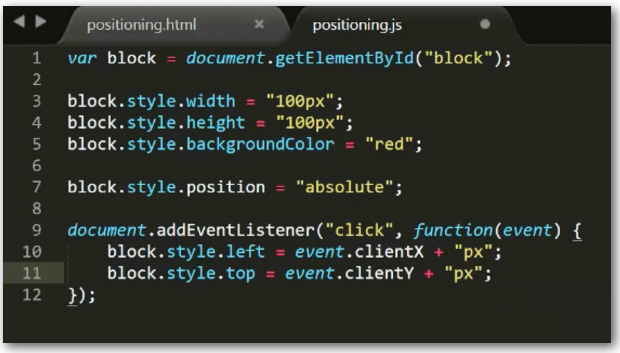

In positioning.js, let's get a reference to that div, using get element by ID block. Now, let's change the style of that div, to adjust its width and height, and change its background color, so that we can see it. Of course, you could just as easily do this with CSS, but it's good to get practice setting styles with JavaScript; not that you'll replace using CSS files with code, but you'll be doing it often enough that you should get used to it.



So, there we can see that div sitting up there at the top of the page. Now, by default, positioning of HTML objects is automatic, based on the layout of the page. In order to specify exact positions of elements, we'll have to change the element's position style to either absolute, or relative. I'll use absolute. Next, listen for the click event on the document itself. Remember that, a mouse event will have the location of the click in the client X, and client Y properties of that event object.

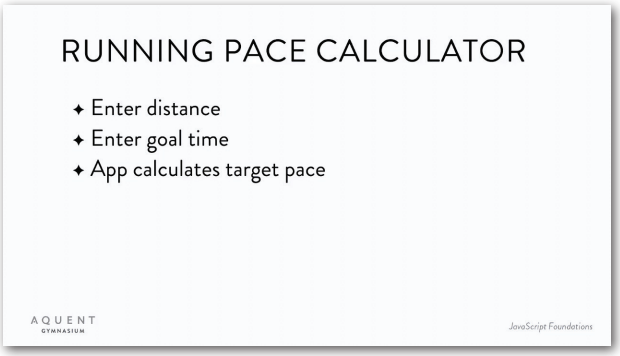
So, we can use those numbers to set the position of this div by saying, block style left equals event.clientX plus px. And, block.style.top equals event.clientY plus px. Note that, we have to add the units, pixels, just like any other CSS numeric property. Now, open that up in a browser, and click around. And, sure enough, the div moves to the exact position of the mouse click.



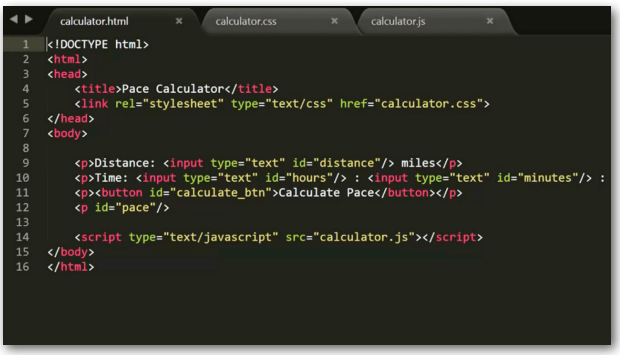

So, now, you know how to select elements, change their content with inner text, or even add new HTML to them with inner HTML, and change their styles; not just statically, but dynamically to reflect a user's actions. You can now begin to make sites and applications that don't just statically sit there, but interact with the user. Next up, we'll attempt to make an actual functioning HTML application. Think you're ready? Let's see.

## CODING AN APPLICATION

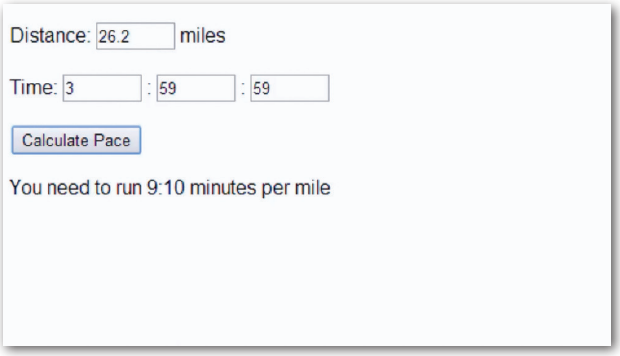
Okay, we finally have all the tools we need to make a real, functioning, dynamic HTML application. I wanted to use some kind of calculator, but financial calculators can be boring. I'm a runner, so I decided to make a running pace calculator. You enter in the distance you want to go, and the time you want to finish in, and it will tell you how fast you need to run. Very useful for planning out a race. If you're not a runner, you'll still learn something here. And, you should be able to easily adapt this to some other kind of calculator, like a miles per gallon fuel calculator, or if you really want, some kind of financial calculator.



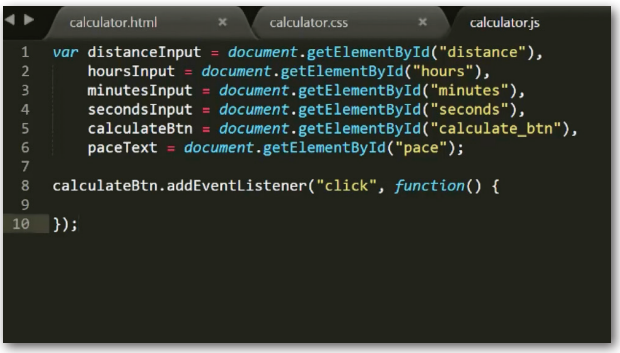

We'll start with this HTML file. You can download the files for this application on the course's web page. This will be calculator.html. As you can see, it just has a few text input fields with some explanatory text, then a button, and an empty paragraph element that will be used to display the results. Note, that all of these elements have IDs for easy access in JavaScript.




Now, the HTML file loads a calculator.js JavaScript file, which is empty right now. And, there are some very basic styles in calculator.css. Here is what this looks like in a browser. The idea is, you enter in the distance you're running, such as five, for a five mile run, or a 13.1 for a half marathon or 26.2 for a full marathon. Then, you enter in the time you want to finish in, in hours, minutes, and seconds. Someday, I'd like to beat four hours in a marathon so I'll enter three, 59, 59. Then, I can click on the Calculate Pace button, and it will tell me how fast to run, to achieve that time. But, right now it's going to do nothing, because we have to write all the code to make that happen.



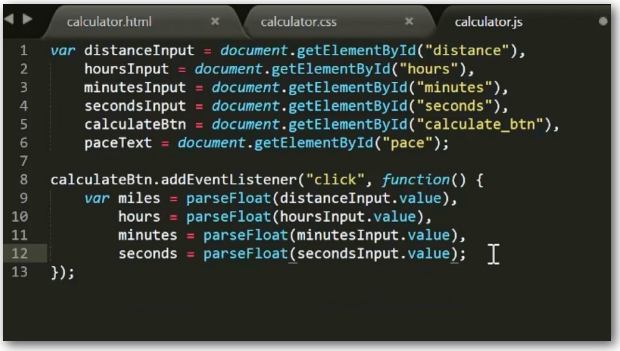

So, let's jump over into the JavaScript. What to do first? Well, I'm going to start by selecting all those elements, using getElementByID, and storing each one in its own variable, so, I can get access to them anytime I want. So, I have distance input, and hours, minutes, and seconds inputs, and the calculate button, and paceText for the result.




Now, JavaScript can just sit back, and wait for the user to enter the values in the inputs, and then click the button. So, we'll listen for a click event on the calculate button. In the handler function for that event, we can now grab all the values from those inputs, and store them in variables.

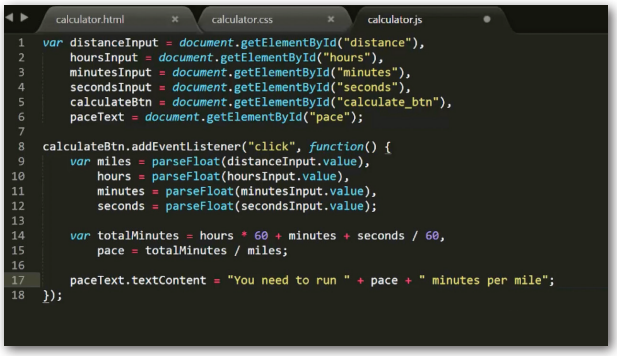
Input elements like this have a value property. This will contain the value entered in that input. So, we can get miles, hours, minutes, and seconds.

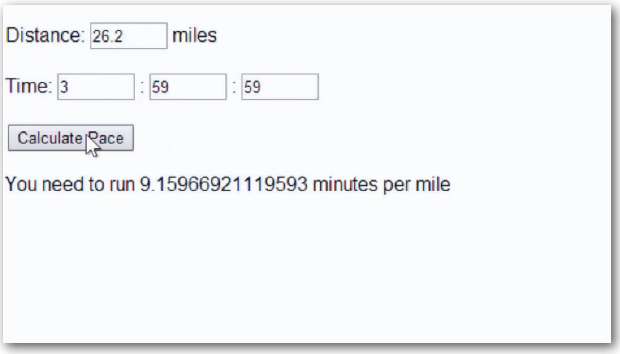
But, those values will come in as strings, not numbers. We could try and rely on type coercion to convert them to numbers, but it's always safer to explicitly convert a value to the type you want. So, what I'm going to do is wrap each one of those in a built-in function called parsefloat. This function takes a string, and converts it into a decimal number, if at all possible. Now, you might be wondering how you can learn about all these useful little functions, like parsefloat. Mostly, it's just experiencing, continually seeking out more information on the language. Again, I'll refer you to the Mozilla developer network site, which has a wealth of information, and resources on learning JavaScript.



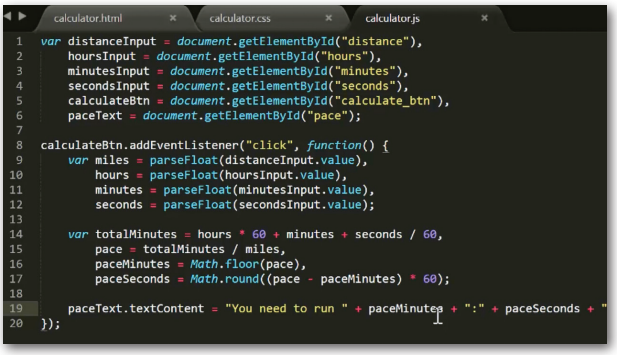

Now, runners generally gauge speed in terms of minutes per mile, rather than miles per hour. So, an elite marathoner might do something around five minutes per mile, whereas someone like me is more like nine or 10. So to do that calculation, we'll need to get those three time variables into a totalMinutes variable. We do that by multiplying hours times 60, adding the minutes variable, and then, adding seconds divided by 60. So, for example, one hour, 30 minutes, and 30 seconds would wind up as 90.5 minutes. Finally, that pace in minutes per mile is the total minutes, divided by the miles. And, we can just set paceText to a string that contains that value.

So, we run that, and input some values. I'll put in 26.2 again, and 3, 59, 59. I click the button, and see that I need to run a 9.15966 something minutes-per-mile Well, I guess that's technically correct, but it's not very user-friendly. Minutes, and seconds per mile would be much better.




Well, we can get the minutes part by using math, floor. This will round that 9.15 something number down to nine, even. Let's call that paceMinutes. Then, if you subtract paceMinutes from pace, you'll get that fractional 0.15966 part. If we multiply that by 60, we'll get the seconds. We could round that off, as we're not really concerned about fractions of a second. Now, we can alter that paceText string to show paceMinutes, plus a colon, plus paceSeconds. And, I run this



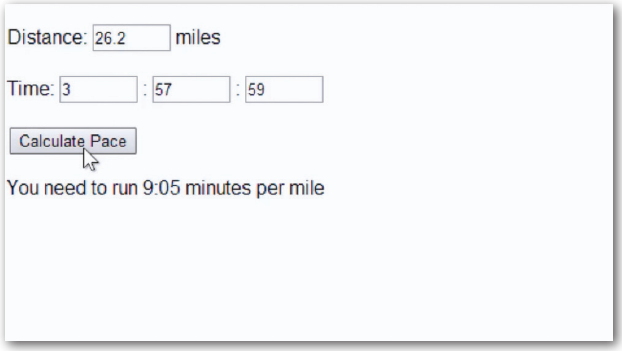
again into my data, and see that I need to run at a pace of 9:10 per mile. That's exactly correct. So, if you've gotten this far, congrats. You've made an actual, useful application.

But, before you start patting yourself on the back too much, let's run this thing through its paces. What if I wanted to knock a couple of minutes off my time, and run a marathon in 3, 57, 59? Well, that pace should be 9:05 per mile, but, hmm, we're missing the zero. Well, welcome to the world of programming. The computer does exactly what you tell it to. It doesn't know that you're formatting a time value. It's just concatenating numbers, and strings that you give it.

So, if paceSeconds is less than 10, we should be adding a zero in there. Well, we can just do that using an if statement. We say, if paceSeconds is less than 10, paceSeconds equals zero plus paceSeconds. Now, note that I used the string zero, not a number. This results in JavaScript coercing paceSeconds into a string to match the string zero. This will result in a string of zero five, in the last example. If I just use the number zero, I'd just be adding the number five to the number zero, which would equal the number five and wouldn't change anything. So, let's try that again. Enter 3, 57, 59, and we have a time of 9:05. Great.

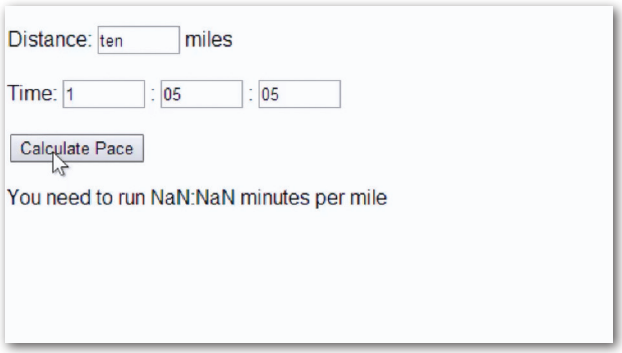
calculator.html | calculator.css | calculator.js

```
var distanceInput = document.getElementById("distance"),
    hoursInput = document.getElementById("hours"),
    minutesInput = document.getElementById("minutes"),
    secondsInput = document.getElementById("seconds"),
    calculateBtn = document.getElementById("calculate_btn"),
    paceText = document.getElementById("pace");

calculateBtn.addEventListener("click", function() {
    var miles = parseFloat(distanceInput.value),
        hours = parseFloat(hoursInput.value),
        minutes = parseFloat(minutesInput.value),
        seconds = parseFloat(secondsInput.value);

    var totalMinutes = hours * 60 + minutes + seconds / 60,
        pace = totalMinutes / miles,
        paceMinutes = Math.floor(pace),
        paceSeconds = Math.round((pace - paceMinutes) * 60);

    if(paceSeconds < 10) {
        paceSeconds = "0" + paceSeconds;
    }
```




Now, this all works perfectly, assuming that our users are perfectionists, and would never do anything so ridiculous, as to forget to enter a number in one of the fields, or worse, enter a non-number into one of the fields. But unfortunately...

Anyway, let's see what would happen if a user did such a thing. I'll enter the word "ten" here, instead of the number for miles. And, I'll enter some random time, and click. And, apparently I need to run a pace of NaN:NaN. What the heck is that all about? Well, this N-A-N is actually another data type I haven't mentioned yet. It stands for, "not a number." It sounds crazy, but that's what you get when you try to do a mathematical operation that JavaScript can't make sense of. It's kind of like "undefined", but only applies to numbers.




Here, this is occurring when I call parseFloat with the string T-E-N-10. ParseFloat can't convert that string to a number, so, it returns NaN. Then, every other calculation that tries to use NaN, results in NaN itself. So, before we use any of these so-called numbers in any calculation, we need to check that they are actually valid numbers, and not not-a-number.

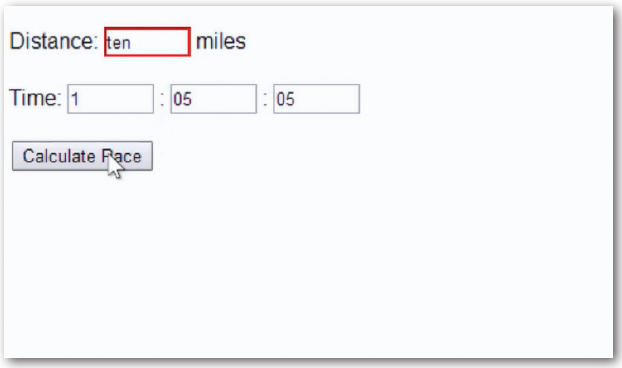
Now, the only way to test for not-a-number is a special function called “isNaN.” So, for miles, we say, if isNaN miles. If this results in true, than there’s some user input error. The user either left a field blank, or entered a non-number. So, we can use some CSS styling to call out that error. We’ll add a red border around that text input. And, since no other calculations will make sense once we have NaN, we can just call return, to exit right out of this function. Earlier, we used return to pass a value back out of the function, but you can use return by itself to just exit the function, right then, and there.

```
calculator.html | calculator.css | calculator.js

var distanceInput = document.getElementById("distance"),
    hoursInput = document.getElementById("hours"),
    minutesInput = document.getElementById("minutes"),
    secondsInput = document.getElementById("seconds"),
    calculateBtn = document.getElementById("calculate_btn"),
    paceText = document.getElementById("pace");

calculateBtn.addEventListener("click", function() {
    var miles = parseFloat(distanceInput.value),
        hours = parseFloat(hoursInput.value),
        minutes = parseFloat(minutesInput.value),
        seconds = parseFloat(secondsInput.value);

    if(isNaN(miles)) {
        distanceInput.style.borderColor = "red";
        return;
    }

    var totalMinutes = hours * 60 + minutes + seconds / 60,
        pace = totalMinutes / miles,
```

Now, if the isNaN check is false, we want to make sure that we set the border color back to its initial value. We can do that by setting border color to the string initial. And, I’m just going to do the exact same thing for all those other inputs. Now, if you’re thinking, we’re doing the same thing, over and over. This would be a good place for a function, you’re exactly right. This would be a great place for a function. And, in fact, that’s going to be one of your assignments for this lesson. So, be thinking about it. But, as is, this should still work, so let’s try it.

I’ll enter a string for miles, and it highlights the error. If I fix that string to a number, and we’re good. Leave out a number for one of these other fields, and again, the error is highlighted. Fix it, and we’re back in business.

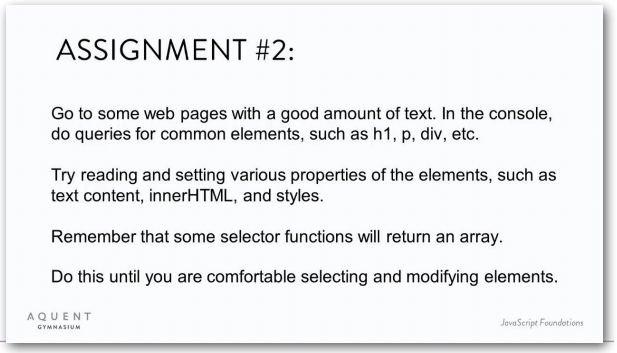
```
calculator.html | calculator.css | calculator.js

var distanceInput = document.getElementById("distance"),
    hoursInput = document.getElementById("hours"),
    minutesInput = document.getElementById("minutes"),
    secondsInput = document.getElementById("seconds"),
    calculateBtn = document.getElementById("calculate_btn"),
    paceText = document.getElementById("pace");

calculateBtn.addEventListener("click", function() {
    var miles = parseFloat(distanceInput.value),
        hours = parseFloat(hoursInput.value),
        minutes = parseFloat(minutesInput.value),
        seconds = parseFloat(secondsInput.value);

    if(isNaN(miles)) {
        distanceInput.style.borderColor = "red";
        return;
    }
    else {
        distanceInput.style.borderColor = "initial";
    }
```




So, if you’ve stuck through the course this far, and you really understand what’s going on in this little application, you really can pat yourself on the back, now. You’re well on your way to becoming a serious developer. More complex applications are just a lot more of this kind of stuff: reading inputs, validating those inputs, processing them one way or the other, and outputting the results.

And, this brings us to the end of Lesson Three. Now, before you go on to Lesson Four, you have some assignments to do. First of all, take the quiz. You’re familiar with this already.

Secondly, go to two or three pages of your choice, preferably ones with a good amount of complex text, open the console, and do several queries for common elements such as h1, p, div, et cetera. Try reading, and setting various properties of elements you get, such as text content, innerHTML, or any styles you want to try. Remember, that some selector func-



tions will give you back an array of items, so you'll have to choose which element to work with, by specifying an index. Do this until you are comfortable selecting different types of elements, and modifying them.

Number three, in the sample application we created, we're validating each numeric property and setting the border style of the input related with it, like so. Try creating a new function in that project called "validateinput." It should take a number value in a text input element as parameters, like so. Now, fill in the body of that function, so that it performs the same action as the original code. Replace each original if/else statement with a call to that new function, like so. See how this makes the overall code smaller, and easier to read.

Four, for an extra challenge, have that validateinput function return true or false, based on whether or not the input is valid. Use that in the original code to jump out of the click event handler function, if the validation fails. This might look something like this. But, there is an even more concise way to do it using the not operator, which is the exclamation point.

In Lesson Four, we'll be doing a lot more with the dom, including creating elements, adding and removing elements from the page, and even looking at a JavaScript templating system to generate a page, or portion of a page, entirely in JavaScript. See you then.