DOM AND JQUERY

But first, two points of review:

A for loop:

```
for ( let i = 0; i < array.length; i++ ) {
        a[i];
        console.log(i);
        // do something else fancy
}
```

An if/else statement:

```
if( true ) {
        // do something
} else if ( true ) {
        // do a different true thing
} else {
        // do the last thing
}
```

With those two pieces of review in mind, a warm up challenge.

FIZZBUZZ

1. Write a short program that prints each number from 1 to 100 on a new line.
2. For each multiple of 3, print "Fizz" instead of the number.
3. For each multiple of 5, print "Buzz" instead of the number.
4. For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number.

Okay, moving on...

So, full disclosure, I debated going futher into **JavaScript** objects before the **DOM** and **jQuery**, but decided against it.

That being the case, things may become more clear as we go deeper in the concept of **objects**.

DOM: Document Object Model

The **DOM** is a programming interface for **HTML**.

It's a representation in **JavaScript** of the structure and content of an **HTML tree**.

HTML tree:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
        <head>
                <meta charset="utf-8">
                <title></title>
        </head>
        <body>
                <h1>HTML Tree</h1>
                <p>A paragraph.</p>
        </body>
</html>
```

<p> is an element node, 'A parapraph.' is a text node.

The **DOM** is a large **data structure** that is exposed to **JavaScript** as an **object**, named *document*, that we can access.

The **DOM** for an **HTML document** is only available in the browser.

Why have the **DOM**?

We can manipulate the **DOM** by accessing the **document object** and all of its child **objects** and **arrays**.

The purpose of the **DOM** is to give us a hook so we can manipulate the contents of our page using **JavaScript**.

So, we can dynamically change text, styling, and remove elements, make things clickable, etc.

So let's see in action. Go to the NY Times website. Open developer tools in Chrome and navigate to the **Console** tab. (CMD + option + j)

- Type **document** into the console and hit enter
- Type **document.** and scroll through the autocomplete suggestions
- Type **docuemnt.head** and hit enter
- **document.body**
- **document.scripts**
- **document.styleSheets**

Note: you can clear the console by typing **clear()** and pressing enter.

You can also run **JavaScript** in the **browser**, which we just alluded to using **document.scripts**.

There are three ways to do this, some of which we've done already in previous demos:

- inline on an element
- inside script tags
- in a separate .js file connected using a script tag

An example of running **JavaScript** inline:

```
<body onload="window.alert('Hello World!')">
```

Note, we did this for demo purposes, in the real world you should never do this. The code will become incomprehensible and impossible to organize.

Running JavaScript in script tags:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
        <head>
                <meta charset="utf-8">
                <title>Script Tags</title>
                <script type="text/javascript">
                        alert('Welcome!');
                </script>
        </head>
        <body>
        </body>
</html>
```

Again, not recommended outside of demo purposes.
But also, sometimes third party scripts require this.

JavaScript from a separate file:

```
<body>
        <script src="js/jquery.js"></script>
        <script src="js/app.js"></script>
</body>
```

Our **scripts**, titled **jquery.js** and **app.js** respectively, are in a **directory** called 'js' at the same level as our **html** file.

As we've mentioned in the past, something we need to be cognizant is that **JavaScript** can start running and try to access **DOM elements** before they've been created.

To avoid this, we put our script tags at the bottom of the page and/or wrap all of our code that access the **DOM** under **window.onload**

```
window.onload = function() {
        alert('page is loaded!');
        // DOM-related code
}
```

This something you'll want to get in the habit of doing to avoid issues. We can only manipulate the **DOM** once our content has been loaded.

Explaining **window.onload**:

We're attaching a **function reference** that's called when the **window's load event** fires.

The **load event** first at the end of the **document** loading process, which includes things like objects, images, scripts, stylesheets, etc.

And remember that **window** is our **global object** in the browser.

It's also helpful to think of JavaScript in three distinct categories:

- Core JavaScript
- DOM API
- Third-party libraries

(API = Application Programming Interface)

Core JavaScript consists of data types, control structures, the standard library, etc. This is what we've been doing thus far, we've really only dealt with **core JavaScript**

Most laptops these days come pre-built with JavaScript engines.

The **DOM API** includes **DOM** objects like **document** and their associated methods, like **.getElementById**.

We get the **DOM API** for free when we run JavaScript in a browser.

A **third party library** is any JavaScript that's included as an extra file, for example **jQuery**. Sometihng like that we have to include ourself.

When we refer to **vanilla JavaScript** it's everything but the **third party libraries**.

Anything we can do with a **third party library**, we can do using vanilla **JavaScript**.

Alright, let's take a moment to set up a folder that contains an **index.html** file, a js folder, and an **app.js** file within the js folder.

Set up the **HTML** file like we always do.

Be sure to link the **app.js** file to your **index.html** file.

Now, in your **app.js** file, put in an alert and make sure it fires when you load your page:

```
alert("Welcome to my application!");
```

If it doesn't fire, or doesn't make sense, look at old slides, Google, ask a neighbor, etc.

From within the app.js file, let's create some new elements:

```
let mainHeading = document.createElement("h1");
let headingText = document.createTextNode("Hello world!");
```

Note, we've only created them but not added them to the **DOM**. Elements and text nodes are only in memory when we create them, we need to explicitly put them onto the page.

This is a good thing, because modifying **DOM element** is costly in terms of load times, so we want to batch them.

To do this, use **.appendChild()**, which works on elements to attach the text node to the element and the element to the page:

```
mainHeading.appendChild(headingText);
document.body.appendChild(mainHeading);
```

To reiterate, steps in adding text to the page:

1. Create a new **h1 element** using the **.createElement() method**
2. Create text using the **.createTextNode() method**
3. Attach the text to the newly created **h1 element**
4. Attach the newly created **h1 element** to the **body element**

We've used **document.getElementById()** to retrieve an element:

```
<ul>
        <li id="happy">Retrieve me!</li>
        <li id="sad">Leave me alone!</li>
</ul>
```

```
let happyListItem = document.getElementById('happy');
console.log(happyListItem)''
```

That's sort of the old school way of retrieving elements, but it's still pertinent and useful. For accessibility reasons, we need these methods as fallbacks.

Other examples would be getElementsByClassName(), querySelector(), and querySelectorAll().

We can use **event listeners**, also called **event handlers**, to designate certain code to run based on **events**.

There are built-in **events** for things that happen on our page, e.g. click, hover, submit, page load, etc.

You find a full list on MDN:
https://developer.mozilla.org/en-US/docs/Web/Events

There are also **global event handlers** available to us in the browser:

- **onclick**
- **onload**
- **onsubmit**

When listening for events using JavaScript, use addEventListener()

We can attach a **listener**, a function to run when the **event** occurs, to an **element** on the page.

```
element.addEventListener(type, listener);
```

We can retrieve the **element** using the methods mentioned previously.

```
<a href="https://google.com/">Click me!</a>
```

```
let anchorElement = document.querySelector('a');
anchorElement.addEventListener('click', event => {
        event.preventDefault();
        console.log('You clicked the anchor tag!');
});
```

INTRO TO JQUERY

So, everything we just did with **vanilla JavaScript** we can do with **jQuery**, a **third party library**, and often times more efficiently.

If **jQuery** is easier why did we do everything with **vanilla JavaScript?**

It's important to understand that **jQuery** uses **vanilla JavaScript** and is written with **vanilla JavaScript**.

You might not always have access to or be able to use **jQuery**. For example, React (a third party library) discourages this.

Some **jQuery** stats and upsides:

- Released in 2006 by John Resig
- Open source, extensively tested
- Smooths conflicts between browsers
- ~18% of all websites on teh internet use it
- ~78% of the top million sites use it
- (Those stats may have changed...)

Some downsides:

- It's ~11,000 extra lines of code that we need to include
- That's obviously not so great in terms of bandwidth, but this mitigated if we use the minified version
- Provides more than we'll actually use
- Often times, you probably don't even need it

jQuery's primary function is literally called **jQuery**, also aliesed under **$()**

It does different htings depeending on the argument passed to it.

Some frameworks, like WordPress, force you to use **jQuery()** over **$()** to prevent conflicts.

The primary use of the **jQuery function** is to select a group of **elements**, similar to **querySelectAll()**

However, the **jQuery function** returns a **jQuery object**, an array-like collection of elements.

Note, we only have access to **jQuery methods** on **jQuery objects**, not **vanilla JavaScript objects**.

```
<ul class="special-list">
        <li>First child</li>
        <li>Second child</li>
</ul>
```

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js"></s
$(".special-list li")
// Returns a jQuery object containing all the 'li' inside '.sp
```

We can perform many actions on the resulting jQuery object.

Some examples include adding content, changing styles, adding or removing classes, adding event listeners.

We can perform operations all at once on all the elements in the collection, and we don't have to manually iterate over the element.

Finally, don't worry about knowing every single DOM API method and every single jQuery method.

Get comfortable (as always) reading documenation. You'll be reading the JavaScript documenation on MDN and jQuery documenation, and Google, all the time. It's normal.

https://api.jquery.com/