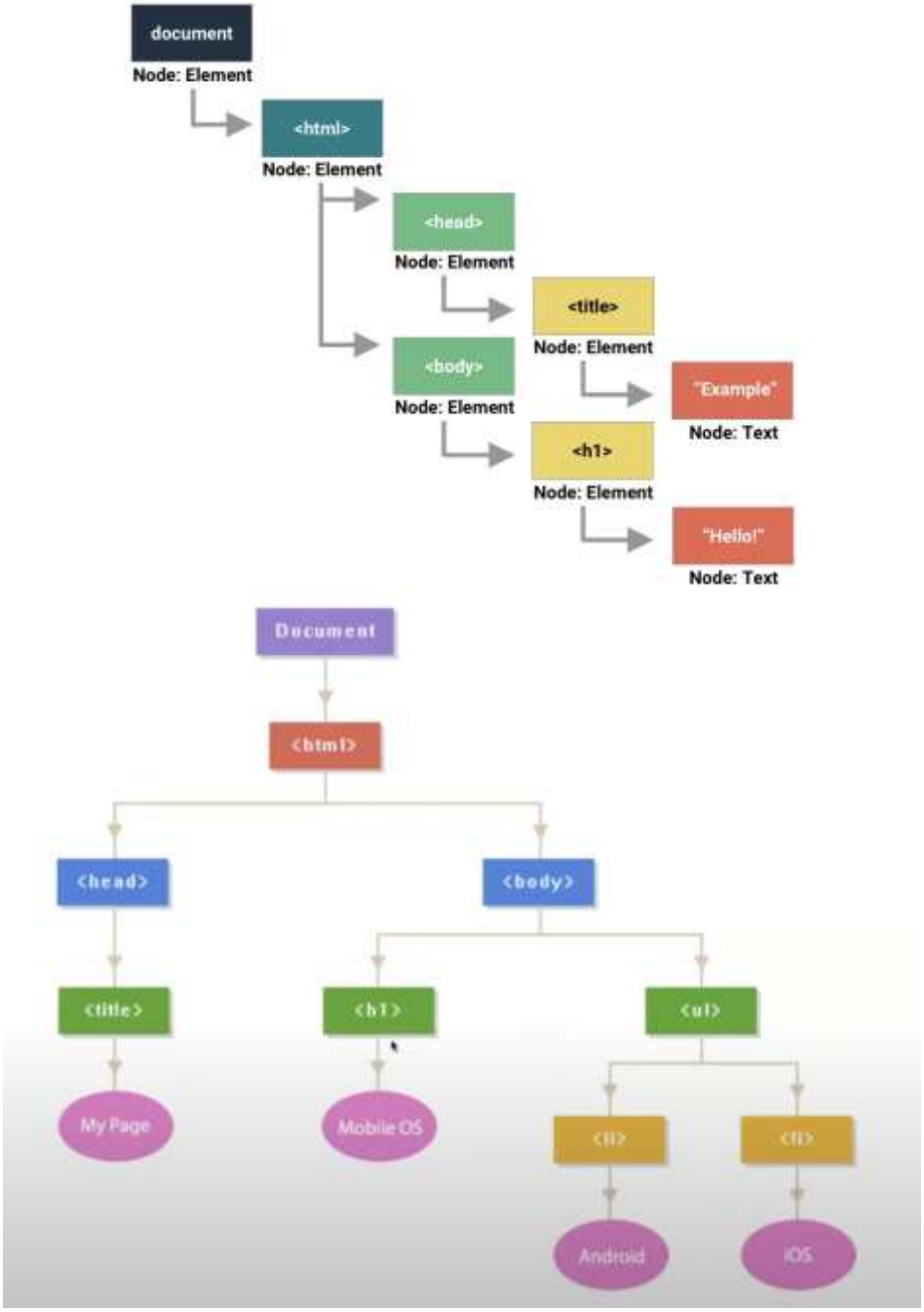
Unit 2 Sprint 1

DOM (Module 1)

DOM

When a web page is loaded into a browser, the browser first looks for the HTML file. The browser uses the HTML file as a blueprint, or instructions on how to build the page (this coupled with the CSS file later). The browser parses these instructions and builds a model for how the page should look and act using JavaScript. This model is a JavaScript Object containing every element in order on the page. This Object is referred to as the DOM, or Document Object Model.
The DOM is built as a data structure known as a 'Tree', because parent elements have nested children elements (or leaves). As with physical trees, we can follow branches of the tree to get to the exact leaf (or leaves) that we want to access. Each branch of our DOM tree can be its own tree.



Selecting

- getElement = (HTMLCollection) These are the original methods for selecting elements from the DOM. They each take a single string as the only argument, containing either the id or class you are looking for.

```
document.getElementsByTagName('p');
```

- querySelector = (NodeList) These methods allow us to select element(s) based on CSS style selectors (remember `.` is class and `#` is id). Each method takes a string containing the selectors and returns the element(s). Note - we can select by element, id, class, or others with both methods.

```
document.querySelector('.custom-style')
```

~ Note: with getElement you specify whether it is a tag name, id or class (getElementsByTagName) so you don't use . or # in the () but with querySelector you don't specify whether it is an id or class so use . or # in the ()

HTML Collection

- Represents a generic collection (array-like object similar to arguments) of elements (in document order) and offers methods and properties for selecting from the list
- Similar to array because both have length property and they have index based items
- Can't use any other array methods

NodeList

- Like HTML Collection but also allows you to use .forEach()

Array.from()

- This is a method that can create an array from an array-like object (html collections and nodelists)
- To use this, give .from the array-like object as its only argument
- Syntax: Array.from(arrayLikeObject)

Manipulating

.textContent

- Can use the assignment operator (=) to change the text of an element
    - `element.textContent = 'Something New';`

.setAttribute()

- This method is used as a way to set or reassign an attribute on the element.
    - Takes two arguments, the attribute to set, and the value to set to that attribute.
    - `element.setAttribute('src', 'http://www.imagsource.com/image.jpg')`

.style

- Every element contains a style object. This property accesses that style object. The style object contains every available style as a key and a value as the value.
- You can access and change a property on the style object by using the assignment operator `=`.
    - `element.style.color = 'blue';`

.className and .id

- `.className` accesses or assigns a string containing all of the classes on the element.
- `.id` accesses or assigns a string containing the id of the element.

.classList

- `classList` will return an array-like object of all the classes on the element. There are many useful methods available on `classList`.
    - `classList` is a `DOMTokenList`.
    - A `DOMTokenList` is an array-like object with a numerical zero-based index, a length property, also the `.contains()` and `.forEach()` methods.
    - Most notably the methods `.add()` `.remove()` and `.toggle()` exist. All three take a single string representing the class.
        - `.add('className')` and `.remove('className')` do as their names indicate.
        - `.toggle('className')` will add the class if it does not exist and remove it if it does.

.appendChild() and .prepend()

- These methods add child elements to parent elements.
- `.appendChild(child)` will take an element and add it to it's children. It will add it to the 'end' physically so if the children are displayed in order it will be the last.
    - `parentElement.appendChild(childElement)`
- `.prepend(child)` adds a child to the beginning, displaying it first.
    - `parentElement.prepend(childElement)`

.children and .parentNode

- These properties are used for accessing relatives of the element.
- `.children` returns an `HTMLCollection` of all the children of that element.
- `.parentNode` returns the parent element of that element.

Creating

.createElement

- `.createElement` creates a brand new element based on a given string.
- New element exists in memory, but not on the DOM yet.
- Can use any DOM property or method to style and manipulate the element.
    - `document.createElement('h1')` will create an `h1` element.

Appending

.appendChild() and .prepend()

- Add child elements to parent elements.
- `.appendChild(child)` add an element to it's children. Adds to the 'end', so that if displayed in order, the added child will be last.
    - `parentElement.appendChild(childElement)`
- `.prepend(child)` adds a child to the beginning, displaying it first.
    - `parentElement.prepend(childElement)`

Misc:

:nth-of-type(3) = selects the 3rd item, :nth-of-type(5) selects the 5th item, etc

.nextElementSibling = selects the next item in the div

insertAdjacentHTML = does what append and prepend do but you can specify where it goes, not just at the beginning or the end

DOM 1 (Module 2)

Events

- Every user interaction with a site is an event: a click, moving the mouse, scrolling the page, pressing a key on the keyboard, these are all events on the page, and the browser can detect all of them
- When an event happens on a page, it is known as a **trigger**

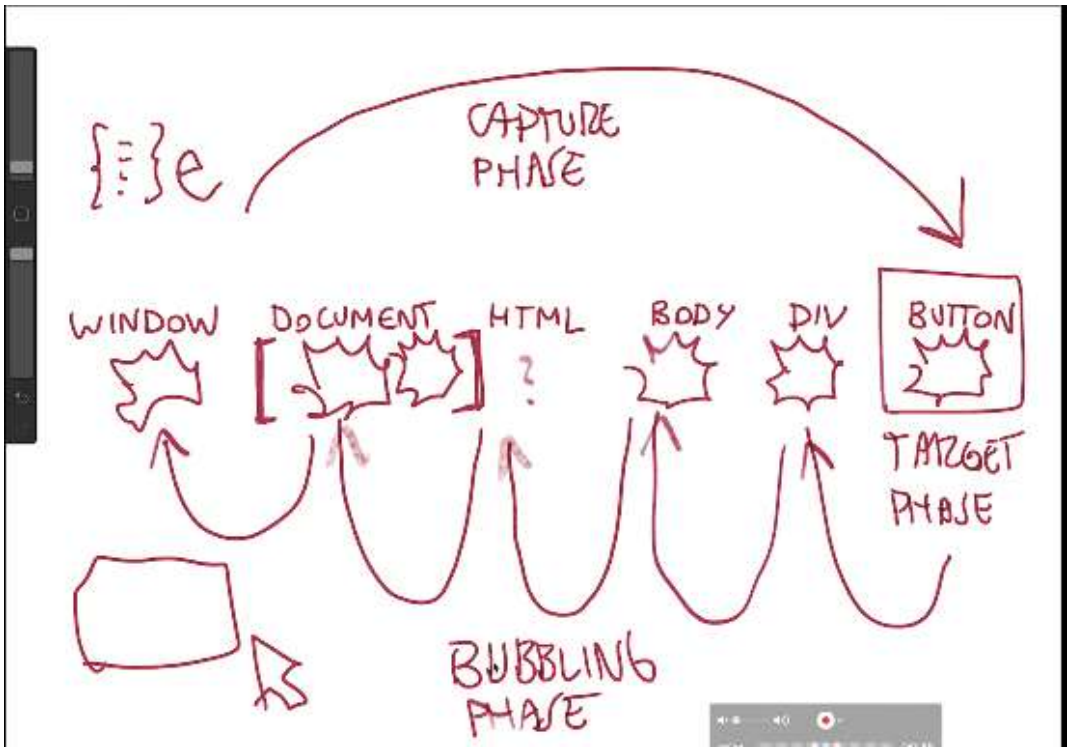
Event Listener

- The browser is always tracking events; also needs to listen for specific events on specific elements
- The tracking process and subsequent action taken is called an event listener
- We put an event listener on an element and give it a callback which is run when the event is triggered

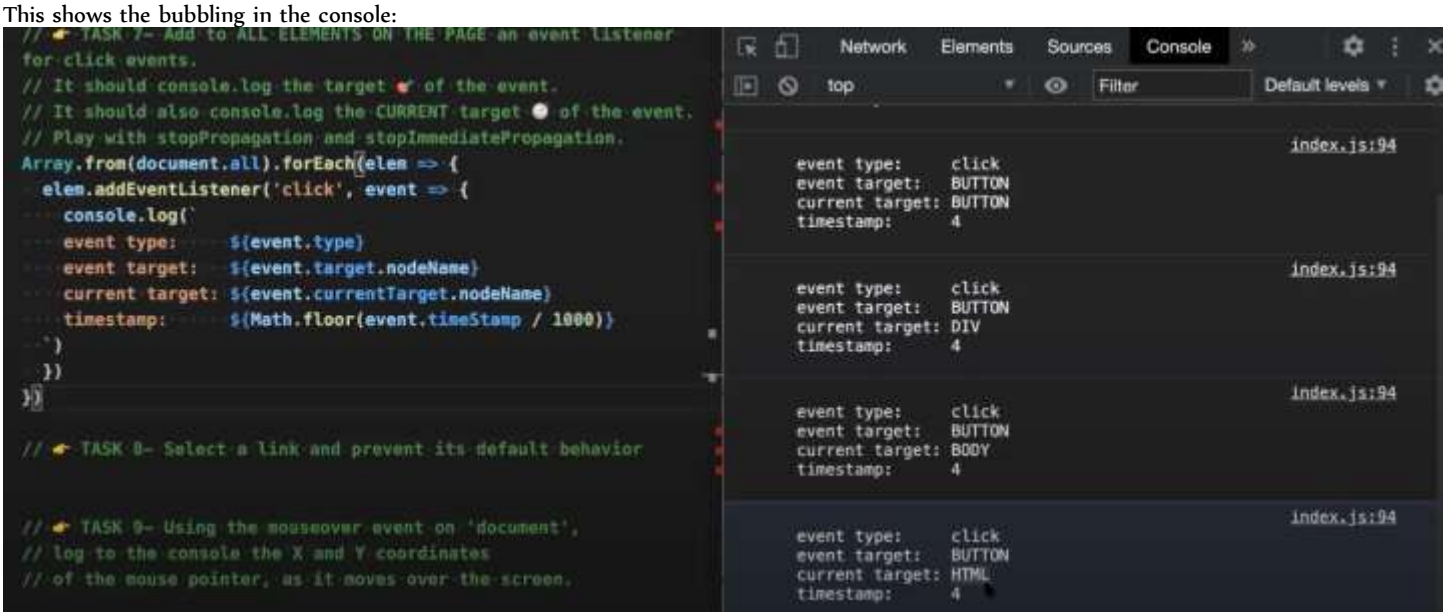
.addEventListener

- Takes two arguments: 1) **Event** to listen for 2) **Callback** to fire when that event is triggered
  `element.addEventListener('click', callback);`
    - **Callback** (aka **Event Handler**) will take a single argument, known as the **Event Object**
    - `element.addEventListener('click', (event) => { event.target.style.backgroundColor = 'blue'; });`

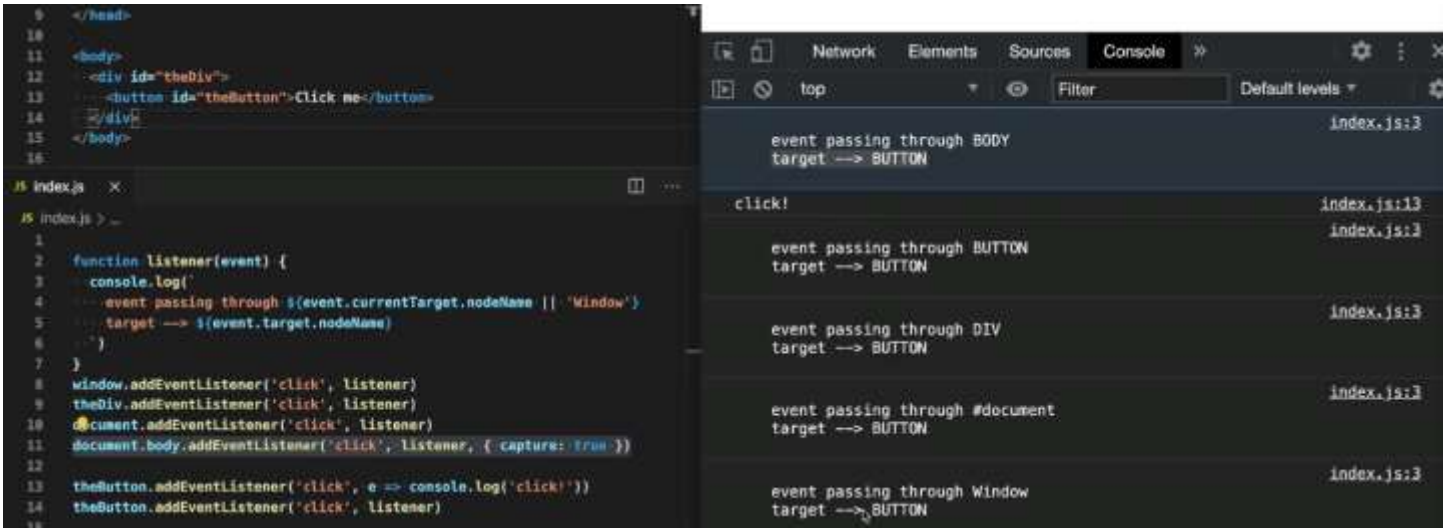
The Bubble Phase:



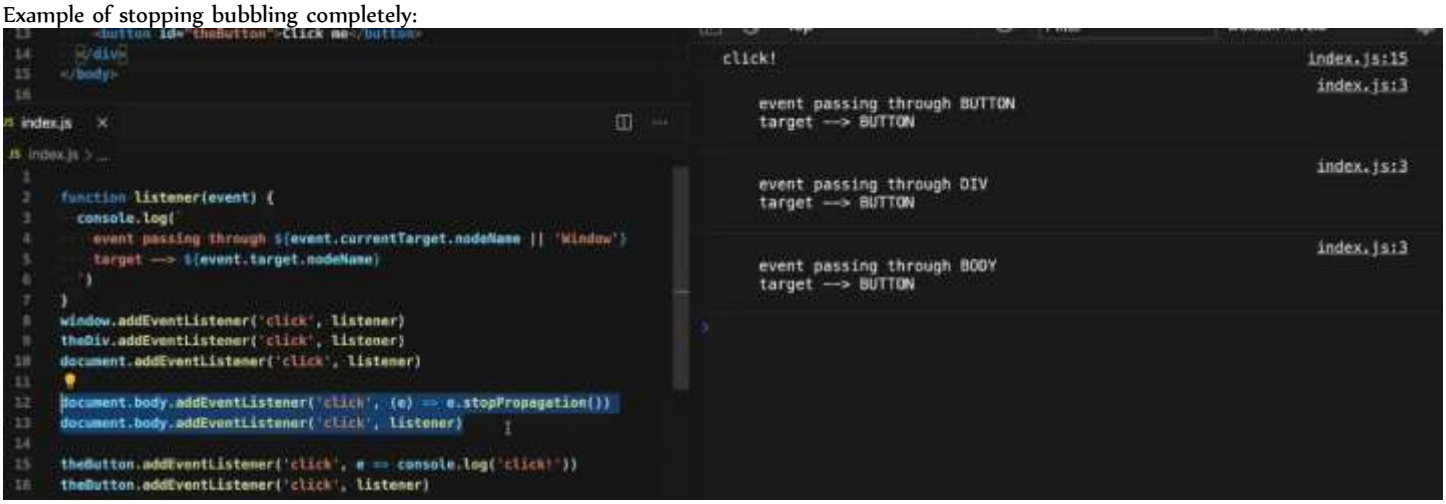
This shows the bubbling in the console:



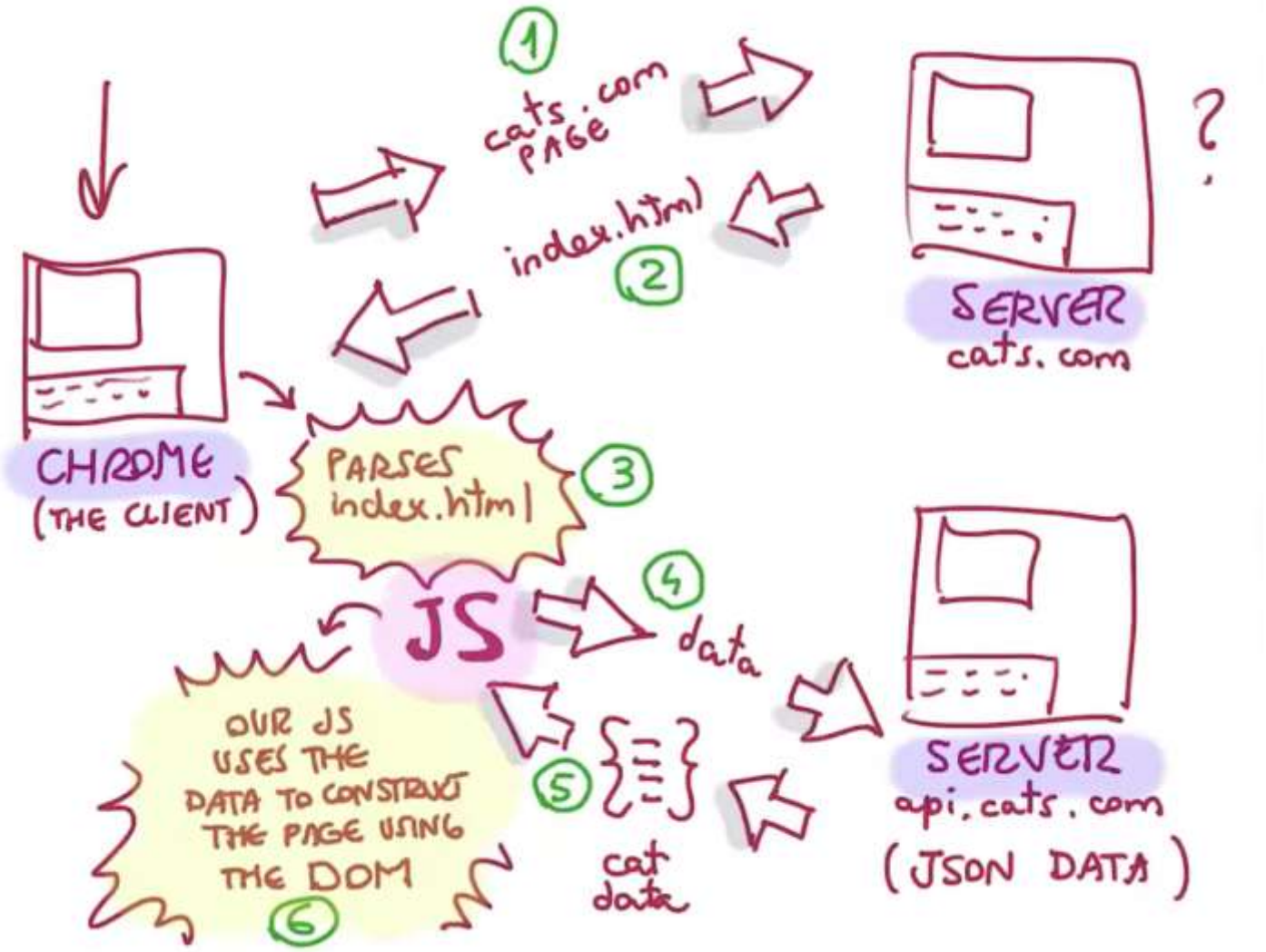
Example of firing a particular node before it would normally happen with bubbling:



Example of stopping bubbling completely:



This shows the flow of requests:



Destructuring

- Does in one line what would have taken two lines to do

```
// const open = constants.open
// const close = constants.close
const { open, close } = constants // destructuring
```

Components 1 (Module 3)

Components

Components

- Dictionary: a part or element of a larger whole
- Single Modular Peices of code
- Usually consists of HTML, CSS, and JS
- Reusable
- D.R.Y.
- Can Stand Alone

Components – CSS

- Components should be modular or stand-alone. With that in mind, you should try to think of your component CSS in a way that could be moved around at any moment and not reliant on any other styles being in place

Components – JavaScript

- JavaScript is used to consume the data and output the content into the DOM. JavaScript's involvement in components is the glue that ties everything together. We can use Javascript to consume the HTML and return a component version of it

Components – Functions

- An example of how functions are useful is when thinking about creating multiple similar buttons. You would create a function that is a buttonCreator which would create a button with the specifications you choose. In order to keep from creating the same exact button you would use parameters (what we later learn as props) to pass info into the function:

```
function buttonCreator(buttonText){
    const button = document.createElement('button');

    button.textContent = buttonText;

    button.classList.add('button');

    button.addEventListener('click', (e) => {
        console.log('clicked!');
    });

    return button;
}

let firstButton = buttonCreator('Button 1');

let secondButton = buttonCreator('Button 2');

parent.appendChild(firstButton);
parent.appendChild(secondButton);
```

.forEach

- runs the array through a loop, passing each item to our callback function. It doesn't return a new array or mutate the data at all

.map

- returns a new array with the items transformed (by our callback)
- Now that we have an array of DOM elements (components), we can do whatever we'd like with them

Components II (Module 4)

Asynchronous code

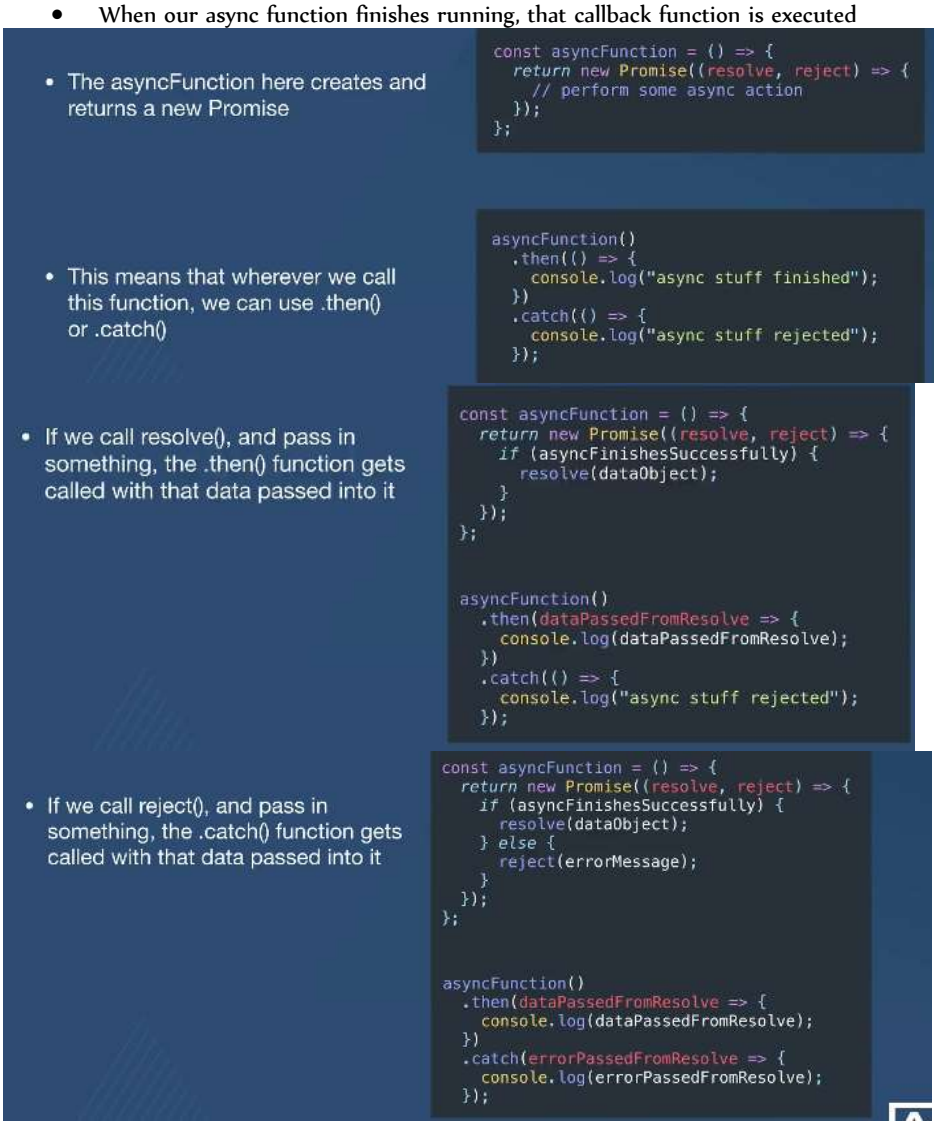
- It is what makes it possible for a JavaScript engine to do two things at a time (asynchronously)
- Use async code to allow the browser to keep executing code while something else is happening. Usually a call to an external API for data
- When we use this technique, we create a helper object, a **Promise**, to inform the browser that the second async task is finished

Promise

- It is an object with a few properties
- When we want to run some async code, we create a new Promise, and use that Promise to inform the JavaScript engine that the async function has finished
- When we instantiate a new Promise with the "new" keyword, we pass in a callback function that receives a "resolve" function and a "reject" function
- If the async function finishes and was successful, we call the resolve() function. If it was unsuccessful, we call the reject() function
- When a Promise is resolved or rejected, we use the Promise object's methods .then() or .catch() to tell the JavaScript engine what to do next

.then() and .catch()

- These are both methods on the Promise object that receive a callback function as an argument
- When our async function finishes running, that callback function is executed



HTTP

- It is a network protocol, a set of rules that govern the way web clients, like a browser, communicate with web servers over the internet
- HTTP Methods ~ provide a common language or nomenclature that the client can use to let the server know what operation it wants to perform
    - When a **client** needs to ask a server for information it should do a **GET** request, specifying a URL that points to the desired resource
    - A **POST** request is used to ask the server to add or create new resources
- HTTP Status Codes ~ used to indicate if a request has been successful or not and why

Axios

- A JavaScript library used to send HTTP requests to servers
- Because all server requests are asynchronous, axios uses Promises
- Inserting axios:

- Inserting axios with script tag:
  <script defer src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
  - You need to place this tag just above the <script defer src='index.js'></script> tag
- Inserting axios using npm:
  npm install axios

  Scaffolding:

```
axios
.get('url')
.then()
.catch()
```

HTTP request

<Start Line

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 155
Content-Type: application/json; charset=utf-8
Date: Thu, 11 Mar 2021 17:50:17 GMT
Etag: W/"9b-/fbzecT8jTKZVDsj1mwIKaAIUOs"
Server: Cowboy
Via: 1.1 vegur
X-Powered-By: Express
```

< Headers

Misc:

Pending vs Fulfilled vs Rejected vs Settled

- Promise is pending before it is fulfilled or rejected
- Promise is fulfilled if the process worked
- Promise is rejected if we get a 404 error code
- Promise is settled when it comes back as fulfilled or rejected

Differences from `Node.appendChild()`:

- `ParentNode.append()` allows you to also append `DOMString` objects, whereas `Node.appendChild()` only accepts `Node` objects.
- `ParentNode.append()` has no return value, whereas `Node.appendChild()` returns the appended `Node` object.
- `ParentNode.append()` can append several nodes and strings, whereas `Node.appendChild()` can only append one node.