JS with DOM

Topics:

1. The Document Object Model(DOM)
2. Creating Content with JavaScript (Manipulation with DOM)
3. Working with browser Events
4. Performance
5. Document Object Model (DOM)

DOM

After receiving the HTML file following steps happen:

* HTML is received
* HTML tags are converted to tokens
* tokens are converted to Nodes
* Nodes are converted to the DOM

the browser has received the bytes that've been sent by a server. The browser has converted the bytes to tags and has read through the tags to create a list of tokens.

This list of tokens then goes through the tree construction stage. The output of this stage is a tree-like structure - this is the DOM!

*DOM: a tree structure that captures the content and properties of the HTML and all the relationships between the nodes*

a JavaScript object is a tree-like structure that has properties and values. So the DOM can be accessed using a special object provided by the browser: document

The document object is provided by the browser and is a representation of the HTML document. This object is not provided by the JavaScript language. ECMAScript is the language specification that JavaScript is based on, and it only references the document object model in one place, in its "Global Object" section

Basically, this says that the document object is not part of JavaScript, but is expected to already exist and be freely accessible to JavaScript code.

Recap:

The DOM stands for "Document Object Model" and is a tree-like structure that is a representation of the HTML document, the relationship between elements, and contains the content and properties of the elements.

The DOM is *not*:

* part of the JavaScript language

The DOM is:

* constructed from the browser
* is globally accessible by JavaScript code using the document object

Selecting Page Element:

.getElementById()

There are a couple of important things to keep in mind about this method:

* it is called on the document object
* it returns a *single* item

.getElementsByClassName()

.getElementsByTagName()

There are a few important things to keep in mind about these two methods:

* both methods use the document object
* both return multiple items
* the list that's returned is not an array

Nodes, Elements, and Interfaces

 The Element Interface inherits all of the Node Interface's properties and methods. This means that any element (lowercase "e"!) that was created from the Element Interface is also a descendent from the Node Interface...which means the element (lowercase "e"!) is also a node (lowercase "n"!).

[Web API Interfaces](https://developer.mozilla.org/en-US/docs/Web/API)

More Ways To Access Elements

.querySelector() - returns a single element (*only return the*first*item it finds.)*

.querySelectorAll() - returns a list of elements

the list returned by .querySelectorAll() is a NodeList. We saw that it is possible to loop over a NodeList with either its .forEach() method, or the humble for loop

1. Manipulation with DOM

Update Existing Page Content

The .innerHTML property sets or returns the HTML content inside the selected element (i.e. between the tags).

There's also the rarely used .outerHTML property. .outerHTML represents the HTML element itself, as well as its children.

The .textContent property will:

* set the text content of an element and all its descendants
* return the text content of an element and all its descendants

.textContentcompletely ignores any CSS styling and returns all of the element's HTML just as it's listed in the HTML. On the other hand, the .innerText property will take CSS styling into consideration and will return the text that is visibly rendered on the page.

Add New Page Content

the .createElement() method is a method on the document object.

The .appendChild() method is called on one element, and is passed the element to append. The element that is about about to be appended is added as the last child.

Just like you created new elements with the .createElement() method, you can also create new text nodes using the .createTextNode() method.

instead of creating a new text node and appending it to an element, it's faster and easier to just update the element's text with the .textContent property.

The .insertAdjacentHTML() method has to be called with two arguments:

* the location of the HTML
* the HTML text that is going to be inserted

Syntax: document.insertAdjacentHTML(position,Text);

*<!-- beforebegin -->*

<p>

*<!-- afterbegin -->*

Existing text/HTML content

*<!-- beforeend -->*

</p>

*<!-- afterend -->*

## Recap:

* .createElement() to create new elements
* .appendChild() to add a child element to a parent element as its last child
* .createTextNode() to create a text node
* .insertAdjacentHTML() to put HTML text anywhere around an element

Some important things to note are:

* if an element *already exists in the DOM* and this element is passed to .appendChild(), the `.appendChild() method will *move it* rather than duplicating it
* an element's .textContent property is used more often than creating a text node with the .createTextNode() method
* the .insertAdjacentHTML() method's second argument has to be text, you can't pass an element

Remove Page Content

.removeChild()

.remove()

The difference is that with .removeChild() must be called on the parent of the element being removed and must be passed the child to be removed, while .remove() can be called directly on the element to delete.

We also learned about the following helpful properties:

* .firstChild
* .firstElementChild
* .parentElement

The difference between .firstChild and .firstElementChild, is that .firstElementChildwill always return the first element, while .firstChild *might* return whitespace (if there is any) to preserve the formatting of the underlying HTML source code.

Style Page Content

.style.<prop> : modifying individual styles

.cssText: The cssText property sets or returns the contents of a style declaration as a string.

.setAttribute():

The setAttribute() method adds the specified attribute to an element, and gives it the specified value.  
If the specified attribute already exists, only the value is set/changed.  
  
**Note:** Although it is possible to add the style attribute with a value to an element with this method, it is recommended that you use [properties of the Style object](https://www.w3schools.com/jsref/dom_obj_style.asp) instead for inline styling, because this will not overwrite other CSS properties that may be specified in the style attribute.

.className:   
  
The .className property returns a space-separated string of the classes.  
  
.className is a property, so we can set its value just by assigning a string to the property:

mainHeading.className = "im-the-new-class";

The above code *erases* any classes that were originally in the element's class attribute and replaces it with the single class im-the-new-class.

.classList :

The classList property returns the class name(s) of an element, as a DOMTokenList object.

The classList property is read-only, however, you can modify it by using following methods:

* .add() - to add a class to the list
* .remove() - to remove a class from the list
* .toggle() - to add the class if it doesn't exists or remove it from the list if it does already exist
* .contains() - returns returns a boolean based on if the class exists in the list or not

1. Working with Browser Events

The monitorEvents() function will keep spitting out all of the events that are happening on the targeted element until the end of time...that, or until you refresh the page. Alternatively, the Chrome browser does offer an unmonitorEvents() function that will turn off the announcing of events for the targeted element.

monitorEvents is that this is for development/testing purposes only. It's not supposed to be used for production code.

Respond to Events



The EventTarget is at the top of the chain. This means that it does not inherit any properties or methods from any other interfaces. However, every other interface inherits from it and therefore contain its properties and methods.

the document object and any DOM element can be an event target. And again, why is this?...because both the Element Interface and the Document Interface inherit from the EventTarget Interface.

EventTarget Interface doesn't have *any* properties and only three methods! These methods are:

* .addEventListener()
* .removeEventListener()
* .dispatchEvent()

## Adding An Event Listener

Using the .addEventListener() method will let us listen for events and respond to them!

pseudo-code to explain how to set an event listener:

<event-target>.addEventListener(<event-to-listen-for>, <function-to-run-when-an-event-happens>);

So an event listener needs three things:

1. an event target - this is called the **target**
2. the type of event to listen for - this is called the **type**
3. a function to run when the event occurs - this is called the **listener**

[list of events](https://developer.mozilla.org/en-US/docs/Web/Events)

Remove an Event Listener

 the .removeEventListener() method requires you to pass the same exact listener function to it as the one you passed to .addEventListener().

pseudo-code

<event-target>.removeEventListener(<event-to-listen-for>, <function-to-remove>);

Remember, the listener function must be the exact same function as the one used in the .addEventListener() call...not just an identical looking function.

Looking into DevTool for EventListener: https://youtu.be/chX2ZNzGXZo

Phases of an Event

Event Phases

There are three different phases during the lifecycle of an event. They are:

* the **capturing** phase
* the **at target** phase
* and the **bubbling** phase

And they actually follow the order above; first, it's *capturing*, then *at target*, and then the *bubbling* phase.

Video: <https://www.youtube.com/watch?v=rGhgPPYc2sg&feature=youtu.be>

Up until this point, we've only seen the .addEventListener() method called with *two* arguments, the:

* event *type*
* and the *listener*

document.addEventListener('click', **function** () {

console.log('The document was clicked');

});

There's actually a third argument to the .addEventListener() method; the useCapture argument. From it's name, you'd think that if this argument were left out, .addEventListener() would default to using the capturing phase. This is an incorrect assumption! **By default, when .addEventListener() is called with only two arguments, the method defaults to using the bubbling phase.**

The code below uses .addEventListener() with only two arguments, so it will invoke the listener during the bubbling phase:

document.addEventListener('click', **function** () {

console.log('The document was clicked');

});

However, in this code, .addEventListener() is called with three arguments with the third argument being true (meaning it should invoke the listener earlier, during the capturing phase!).

document.addEventListener('click', **function** () {

console.log('The document was clicked');

}, true);

Video: <https://www.youtube.com/watch?v=4WYyk3AhCOU&feature=youtu.be>

The Event Object

When an event occurs, the browser includes an **event object**. This is just a regular JavaScript object that includes a ton of information about the event itself.

Up until this point, I've been writing all of the listener functions without any parameter to store this event object. Let's add a parameter so we can store this important information:

document.addEventListener('click', **function** (event) { *// ← the `event` parameter is new!*

console.log('The document was clicked');

});

Notice the new event parameter that's been added to the listener function. Now when the listener function is called, it is able to store the event data that's passed to it!

prevent the default action with .preventDefault()

Avoid Too Many Events

Event Delegation is the process of delegating to a parent element the ability to manage events for child elements. We were able to do this by making use of:

* the event object and its .target property
* the different phases of an event

Link : <https://classroom.udacity.com/courses/ud117/lessons/f270dbcf-eb43-4ce3-b7be-a74d26023496/concepts/85463be2-3206-434e-aa39-4604965daa29>

Know When The DOM Is Ready

## The DOM Is Built Incrementally

when the HTML is received and converted into tokens and built into the document object model, is that this is a sequential process. When the parser gets to a <script> tag, it must wait to download the script file and execute that JavaScript code. This is the important part and the key to why the placement of the JavaScript file matters!

Notice at the bottom of the code that we have so far is a <script> file. This is using inline JavaScript rather than pointing to an external file. The inline file will execute faster because the browser doesn't have to make another network request to fetch the JavaScript file. But the outcome will be exactly the same for both this inline version and if the HTML had linked to an external JavaScript file.

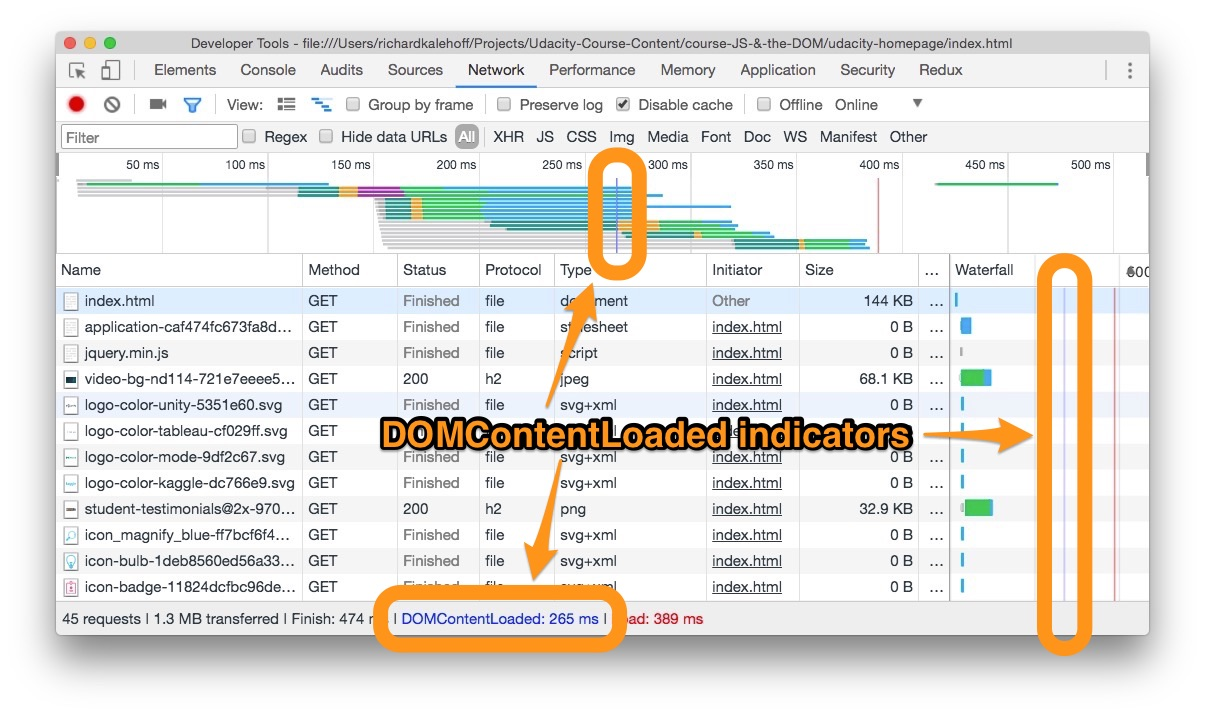
### The Content Is Loaded Event

When the document object model has been fully loaded, the browser will fire an event. This event is called the DOMContentLoaded event, and we can listen for it the same way we listen to any other events:

document.addEventListener('DOMContentLoaded', **function** () {

console.log('the DOM is ready to be interacted with!');

});



1. Performance

Add Page Content Efficiently

Testing Code Performance

performance.now() returns a timestamp that is measured in milliseconds, so it's extremely accurate.

These are the steps to use performance.now() to measure the speed of your code:

1. use performance.now() to get the an initial start time for the code
2. run the code you want to test
3. execute performance.now() to get another time measurement
4. subtract the initial time from the final time

## Using a Document Fragment

using a DocumentFragment to prevent performance issues and to prevent adding unnecessary elements to the DOM.

We can use the .createDocumentFragment() method to create an empty DocumentFragment object.

Reflow & Repaint

**Reflow** is the process of the browser laying out the page. It happens when you first display the DOM (generally after the DOM and CSS have been loaded), and happens again every time something could change the layout. This is a fairly expensive (slow) process.

**Repaint** happens after reflow as the browser draws the new layout to the screen. This is fairly quick, but you still want to limit how often it happens.

If we simply call .removeChild() for each of the two comments that need to be removed, that's one reflow and one repaint for each change (so a total of 2 reflows and 2 repaints). We could rebuild the whole thing in a DocumentFragment and replace #comments -- that's the time to rebuild (possibly involving reading files or data), plus at least one reflow and one repaint.

Or we could hide #comments, delete the spam, and show it again -- that's surprisingly fast, to the cost of one reflow and two repaints (and little else). It's fast because hiding doesn't change the layout, it just erases that section of the screen (1 repaint). When you make the changed section visible again, that's a reflow and a repaint.

In general, if you have to make a group of changes, hide/change all/show is a great pattern to use if the changes are relatively contained.

## Virtual DOM

By the way, this is why React and other "virtual DOM" libraries are so popular. You don't make changes to the DOM, but make changes to another structure (a "virtual DOM") and the library calculates the best way to update the screen to match

## Recap

Reflow is the process of calculating the dimensions and position of page elements. This is a computationally intensive (slow) tasks. Repaint is the process of drawing the pixels to the screen. This is faster than reflow, but is still not a fast process. You want to make sure that your code causes the fewest number of reflows as possible.

The Call Stack

The JavaScript engine keeps a call stack (basically a list) of the functions that are running. When a function is invoked, it is added to the list. When all of the code inside a function has been run, then the function is removed from the call stack. The cool part about a call stack is that a function doesn't have to complete before another function is added to the call stack.

Video: <https://www.youtube.com/watch?v=J9iKeNhoXNM&feature=youtu.be>

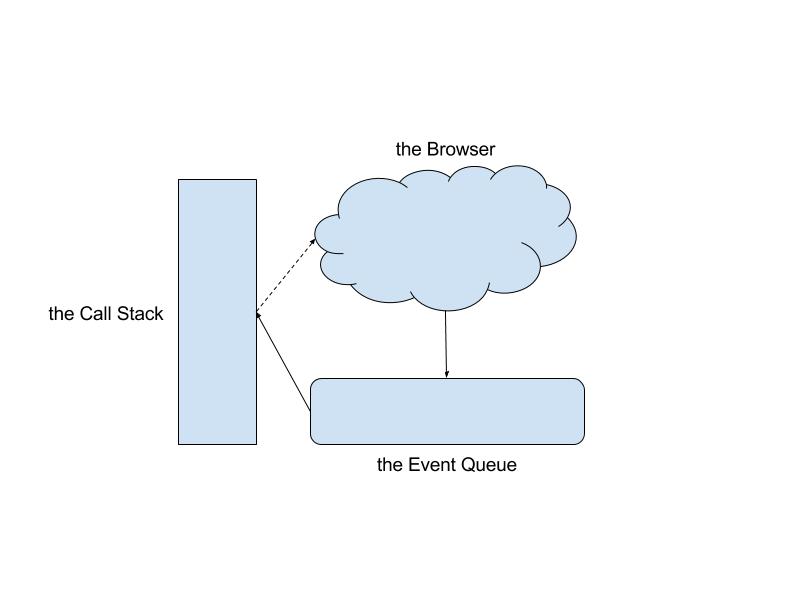
The Event Loop

The simplest explanation of JavaScript's *concurrency model* uses two rules: If some JavaScript is running, let it run until it is finished ("run-to-completion"). If no JavaScript is running, run any pending event handlers.

Since most JavaScript is run in response to an event, this is known as an *event loop*: Pick up the next event, run its handler, and repeat.

There are three parts you have to think about around the event loop:

* the Call Stack
* Web APIs/the browser
* an Event Queue



Not all of the code that we write is 100% JavaScript code. Some of the code is interacting with the Web APIs (also known as "browser APIs"). There are many more examples, but both .addEventListener() and setTimeout() are Web APIs.

console.log('howdy'); *// 1*

document.addEventListener('click', *// 2*

**function** **numbers**() {

console.log('123');

});

console.log('ice cream is tasty'); *// 3*

What happens if someone clicks before this block of code is done? When there is a click event and there is code already running, the numbers function can't just be added directly to the Call Stack because of JavaScript's run-to-completion nature; we can't interrupt any code that might currently be happening. So the function is placed in the Queue. When all of the functions in the Call Stack have finished (also known as idle time), then the Queue is checked to see if anything is waiting. If something is in the Queue, then it's run, creating an entry on the call stack.

**IMPORTANT**: The key things to remember here are 1) current synchronous code runs to completion, and 2) events are processed when the browser isn't busy. Asynchronous code (such as loading an image) runs outside of this loop and sends an event when it is done.

Video: <https://www.youtube.com/watch?v=uBdemYBG-ek&feature=youtu.be>

Asynchronous code makes use of the JavaScript Event Loop. Any asynchronous code (like setTimeout or the function passed to .addEventListener()) is handled by the browser. When this asynchronous code is ready to be executed, it's moved to the queue where it waits until the Call Stack is empty. Whenever the Call Stack is empty, code moves from the Queue over to the Call Stack and gets executed.

setTimeout

## Running Code Later

Similarly to .addEventListener() code being run at some later point, there is the setTimeout()function that will run code at a point later in time. The setTimeout() function takes:

* a function to run at some later time
* the number of milliseconds the code should wait before running the function

Let's check out an example:

setTimeout(**function** **sayHi**() {

console.log('Howdy');

}, 1000);

If we ran this code, the string 'Howdy' would appear in the console in about 1,000 milliseconds or in just about 1 second.

Since setTimeout() is an API provided by the browser, the call to setTimeout() gives the sayHi() function over to the browser which it starts a timer. After the timer is finished, the sayHi() function moves to the Queue. If the Call Stack is empty, then the sayHi() function is moved to the Call Stack and executed.

## setTimeout() with Delay of 0

An interesting aspect to setTimeout() is that we can pass it a delay of 0 milliseconds.

setTimeout(**function** **sayHi**() {

console.log('Howdy');

}, 0); *// ← 0 milliseconds!*

You might think that since it has a delay of 0 milliseconds, that the sayHi function would run immediately. However, it still goes through the JavaScript Event Loop. So the function is handed off to the browser where the browser starts a timer for 0 milliseconds. Since the timer ends immediately, the sayHi function will move to the Queue, and then to the Call Stack once the Call Stack has finished executing any currently-running tasks.

### Break Up Long-Running Code

Do you remember back to a previous section when we wrote JavaScript code to add two hundred paragraphs to the page? Now, instead of adding two hundred paragraphs to the page, what if we added twenty thousand? That's a lot of elements to create, append, and insert into the page!

Now keep in mind how reflow and repaint affect an app's performance. We want to write our JavaScript code to take into consideration reflow and repaint and to cause the fewest number of these.

However, we also want to make sure our app is responsive to user interaction. While JavaScript is running, the page is "busy" and the user won't be able to interact with the page (e.g. clicking a button, filling out a form). Remember that this is because JavaScript runs synchronously. So it will run to completion (creating, appending, and inserting all twenty thousand elements), and it does this before it is able to respond to any actions the user has taken. The function creates all of these elements and adds the to the page will be in the Call Stack until it's completely finished.

One way to give the user a chance to interact with the page is to break up the adding of the content into chunks. Let's do this with setTimeout():

**let** count = 1

**function** **generateParagraphs**() {

**const** fragment = document.createDocumentFragment();

**for** (**let** i = 1; i <= 500; i++) {

**const** newElement = document.createElement('p');

newElement.textContent = 'This is paragraph number ' + count;

count = count + 1;

fragment.appendChild(newElement);

}

document.body.appendChild(fragment);

**if** (count < 20000) {

setTimeout(generateParagraphs, 0);

}

}

generateParagraphs();

This code starts off by setting a count variable to 1. This will keep track of the number of paragraphs that have been added. The generateParagraphs() function will add 500 paragraphs to the page each time it's invoked. The interesting thing is that there's a setTimeout() call at the end of the generateParagraphs() function. If there less than twenty thousand elements, then it setTimeout() will be used to call the generateParagraphs() function.

If you try running this code on a page, you can still interact with the page while the code is running. It doesn't lock up or freeze the page. And it doesn't lock up or freeze because of the setTimeout()calls.

## Recap

The browser-provided setTimeout() function takes another function and a delay, and invokes the function after the delay has passed.

Knowing how the JavaScript Event Loop works, we can use the setTimeout() method to help us write code that allows the browser to handle user interactions.

Properties and Methods

.getElementById();

.getElementsByClassName();

.getElementsByTagName();

.querySelector();

.querySelectorAll();

.innerHTML;

.outerHTML;

.textContent;

.innerText;

.createElement();

.createTextNode();

.appendChild();

.insertAdjacentHTML(position,Text);

.removeChild()

.remove()

.firstChild

.firstElementChild

.parentElement

.style.<prop>;

.cssText();

.setAttribute();

.className;

.classList

• .add() - to add a class to the list

• .remove() - to remove a class from the list

• .toggle() - to add the class if it doesn't exists or remove it from the list if it does already exist

• .contains() - returns returns a boolean based on if the class exists in the list or not;

monitorEvents();

unmonitorEvents();

.addEventListener();

.removeEventListener();

.dispatchEvent();

pseudo-code

<event-target>.addEventListener(<event-to-listen-for>, <function-to-run-when-an-event-happens>);

pseudo-code

<event-target>.removeEventListener(<event-to-listen-for>, <function-to-remove>);

.preventDefault();

DOMContentLoaded event,

performance.now();

.createDocumentFragment();

setTimeout();