### [DOM - Document Object Model](https://www.theodinproject.com/lessons/foundations-dom-manipulation-and-events" \l "dom-document-object-model)

The DOM (or Document Object Model) is a tree-like representation of the contents of a webpage - a tree of “nodes” with different relationships depending on how they’re arranged in the HTML document.

<div id="container">

<div class="display"></div>

<div class="controls"></div>

</div>

In the above example, the <div class="display"></div> is a “child” of <div id="container"></div> and a sibling to <div class="controls"></div>. Think of it like a family tree. <div id="container"></div> is a **parent**, with its **children** on the next level, each on their own “branch”.

### [Targeting Nodes with Selectors](https://www.theodinproject.com/lessons/foundations-dom-manipulation-and-events#targeting-nodes-with-selectors)

When working with the DOM, you use “selectors” to target the nodes you want to work with. You can use a combination of CSS-style selectors and relationship properties to target the nodes you want. Let’s start with CSS-style selectors. In the above example, you could use the following selectors to refer to <div class="display"></div>:

* div.display
* .display
* #container > .display
* div#container > div.display

You can also use relational selectors (i.e. firstElementChild or lastElementChild etc.) with special properties owned by the nodes.

const container = document.querySelector('#container');

// selects the #container div (don't worry about the syntax, we'll get there)

console.dir(container.firstElementChild);

// selects the first child of #container => .display

const controls = document.querySelector('.controls');

// selects the .controls div

console.dir(controls.previousElementSibling);

// selects the prior sibling => .display

So you’re identifying a certain node based on its relationships to the nodes around it.

### [DOM Methods](https://www.theodinproject.com/lessons/foundations-dom-manipulation-and-events#dom-methods)

When your HTML code is parsed by a web browser, it is converted to the DOM as was mentioned above. One of the primary differences is that these nodes are objects that have many properties and methods attached to them. These properties and methods are the primary tools we are going to use to manipulate our webpage with JavaScript. We’ll start with the query selectors - those that help you target nodes.

#### Query Selectors

* element.querySelector(selector) returns a reference to the first match of selector
* element.querySelectorAll(selectors) returns a “nodelist” containing references to all of the matches of the selectors

\*There are several other, more specific queries, that offer potential (marginal) performance benefits, but we won’t be going over them now.

It’s important to note that when using querySelectorAll, the return value is **not** an array. It looks like an array, and it somewhat acts like an array, but it’s really a “nodelist”. The big distinction is that several array methods are missing from nodelists. One solution, if problems arise, is to convert the nodelist into an array. You can do this with Array.from() or the [spread operator.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

#### Element Creation

* document.createElement(tagName, [options]) creates a new element of tag type tagName. [options] in this case means you can add some optional parameters to the function. Don’t worry about these at this point.

const div = document.createElement('div');

This function does NOT put your new element into the DOM - it simply creates it in memory. This is so that you can manipulate the element (by adding styles, classes, ids, text, etc.) before placing it on the page. You can place the element into the DOM with one of the following methods.

#### Append Elements

* parentNode.appendChild(childNode) appends childNode as the last child of parentNode
* parentNode.insertBefore(newNode, referenceNode) inserts newNode into parentNode before referenceNode

#### Remove Elements

* parentNode.removeChild(child) removes child from parentNode on the DOM and returns a reference to child

#### Altering Elements

When you have a reference to an element, you can use that reference to alter the element’s own properties. This allows you to do many useful alterations, like adding/removing and altering attributes, changing classes, adding inline style information and more.

const div = document.createElement('div');

// creates a new div referenced in the variable 'div'

#### Adding Inline Style

div.style.color = 'blue';

// adds the indicated style rule

div.style.cssText = 'color: blue; background: white;';

// adds several style rules

div.setAttribute('style', 'color: blue; background: white;');

// adds several style rules

See DOM Enlightenment’s [section on CSS Style rules](http://domenlightenment.com/" \l "6.2" \t "_blank) for more info on inline styles.

Note that if you’re accessing a kebab-cased CSS rule from JS, you’ll either need to use camelCase or you’ll need to use bracket notation instead of dot notation.

div.style.background-color // doesn't work - attempts to subtract color from div.style.background

div.style.backgroundColor // accesses the div's background-color style

div.style['background-color'] // also works

div.style.cssText = "background-color: white;" // ok in a string

#### Editing Attributes

div.setAttribute('id', 'theDiv');

// if id exists, update it to 'theDiv', else create an id

// with value "theDiv"

div.getAttribute('id');

// returns value of specified attribute, in this case

// "theDiv"

div.removeAttribute('id');

// removes specified attribute

See MDN’s section on [HTML Attributes](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes" \t "_blank) for more info on available attributes.

#### Working with Classes

div.classList.add('new');

// adds class "new" to your new div

div.classList.remove('new');

// removes "new" class from div

div.classList.toggle('active');

// if div doesn't have class "active" then add it, or if

// it does, then remove it

It is often standard (and cleaner) to toggle a CSS style rather than adding and removing inline CSS.

#### Adding Text Content

div.textContent = 'Hello World!'

// creates a text node containing "Hello World!" and

// inserts it in div

#### Adding HTML Content

div.innerHTML = '<span>Hello World!</span>';

// renders the HTML inside div

\*Note that textContent is preferable for adding text, and innerHTML should be used sparingly as it can create security risks if misused. Check out *[this video](https://www.youtube.com/watch?v=ns1LX6mEvyM&ab_channel=WebDevSimplified" \t "_blank)* if you want to see an example of how.

Let’s take a minute to review what we’ve covered and give you a chance to practice this stuff before moving on. Check out this example of creating and appending a DOM element to a webpage.

<!-- your HTML file: -->

<body>

<h1>

THE TITLE OF YOUR WEBPAGE

</h1>

<div id="container"></div>

</body>

// your JavaScript file

const container = document.querySelector('#container');

const content = document.createElement('div');

content.classList.add('content');

content.textContent = 'This is the glorious text-content!';

container.appendChild(content);

In the JavaScript file, first we get a reference to the container div that already exists in our HTML. Then we create a new div and store it in the variable content. We add a class and some text to the content div and finally append that div to container. All in all it’s a simple process. After the JavaScript code is run, our DOM tree will look like this:

<!-- The DOM -->

<body>

<h1>

THE TITLE OF YOUR WEBPAGE

</h1>

<div id="container">

<div class="content">

This is the glorious text-content!

</div>

</div>

</body>

Keep in mind that the JavaScript does not alter your HTML, but the DOM - your HTML file will look the same, but the JavaScript changes what the browser renders.

Your JavaScript, for the most part, is run whenever the JS file is run, or when the script tag is encountered in the HTML. If you are including your JavaScript at the top of your file, many of these DOM manipulation methods will not work because the JS code is being run before the nodes are created in the DOM. The simplest way to fix this is to include your JavaScript at the bottom of your HTML file so that it gets run after the DOM nodes are parsed and created.

Alternatively, you can link the JavaScript file in the <head> of your HTML document. Use the <script> tag with the src attribute containing the path to the JS file, and include the defer keyword to load the file after the HTML is parsed, as such:

<head>

<script src="js-file.js" defer></script>

</head>

Read the second bullet point in [this MDN article](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML" \l "applying_css_and_javascript_to_html" \t "_blank) for more information, which also includes a link to additional script loading strategies.

### [Exercise](https://www.theodinproject.com/lessons/foundations-dom-manipulation-and-events" \l "exercise)

Copy the example above into files on your own computer. To make it work you’ll need to supply the rest of the HTML skeleton and either link your JavaScript file, or put the JavaScript into a script tag on the page. Make sure everything is working before moving on!

Add the following elements to the container using ONLY JavaScript and the DOM methods shown above.

1. a <p> with red text that says “Hey I’m red!”
2. an <h3> with blue text that says “I’m a blue h3!”
3. a <div> with a black border and pink background color with the following elements inside of it:
   1. another <h1> that says “I’m in a div”
   2. a <p> that says “ME TOO!”
   3. Hint for this one: after creating the <div> with createElement, append the <h1> and <p> to it before adding it to the container.

### [Events](https://www.theodinproject.com/lessons/foundations-dom-manipulation-and-events" \l "events)

Now that we have a handle on manipulating the DOM with JavaScript, the next step is learning how to make that happen dynamically, or on demand! Events are how you make that magic happen on your pages. Events are actions that occur on your webpage such as mouse-clicks or keypresses, and using JavaScript we can make our webpage listen and react to these events.

There are three primary ways to go about this: you can attach functions’ attributes directly on your HTML elements, you can set the “on\_event\_” property on the DOM object in your JavaScript, or you can attach event listeners to the nodes in your JavaScript. Event listeners are definitely the preferred method, but you will regularly see the others in use, so we’re going to cover all three.

We’re going to create 3 buttons that all alert “Hello World” when clicked. Try them all out using your own HTML file, or using something like [CodePen](https://codepen.io/" \t "_blank).

#### Method 1

<button onclick="alert('Hello World')">Click Me</button>

This solution is less than ideal because we’re cluttering our HTML with JavaScript. Also, we can only have 1 “onclick” event per element.

#### Method 2

<!-- the HTML file -->

<button id="btn">Click Me</button>

// the JavaScript file

const btn = document.querySelector('#btn');

btn.onclick = () => alert("Hello World");

#### (Need to review **[arrow functions](http://javascript.info/arrow-functions-basics" \t "_blank)**?)

This is a little better. We’ve moved the JS out of the HTML and into a JS file, but we still have the problem that a DOM element can only have 1 “onclick” property.

#### Method 3

<!-- the HTML file -->

<button id="btn">Click Me Too</button>

// the JavaScript file

const btn = document.querySelector('#btn');

btn.addEventListener('click', () => {

alert("Hello World");

});

Now, we maintain separation of concerns, and we also allow multiple event listeners if the need arises. Method 3 is much more flexible and powerful, though it is a bit more complex to set up.

Note that all 3 of these methods can be used with named functions like so:

<!-- the HTML file -->

<!-- METHOD 1 -->

<button onclick="alertFunction()">CLICK ME BABY</button>

// the JavaScript file

function alertFunction() {

alert("YAY! YOU DID IT!");

}

// METHOD 2

btn.onclick = alertFunction;

// METHOD 3

btn.addEventListener('click', alertFunction);

Using named functions can clean up your code considerably, and is a really good idea if the function is something that you are going to want to do in multiple places.

With all three methods we can access more information about the event by passing a parameter to the function that we are calling. Try this out on your own machine:

btn.addEventListener('click', function (e) {

console.log(e);

});

\*Note that *function (e)* is a callback from addEventListener. Further explanation of callbacks can be found *[HERE](https://dev.to/i3uckwheat/understanding-callbacks-2o9e" \t "_blank)*.

The e in that function is an object that references the **event** itself. Within that object you have access to many useful properties and functions such as which mouse button or key was pressed, or information about the event’s **target** - the DOM node that was clicked.

Try this:

btn.addEventListener('click', function (e) {

console.log(e.target);

});

and now this:

btn.addEventListener('click', function (e) {

e.target.style.background = 'blue';

});

Pretty cool, eh?

#### Attaching Listeners to Groups of Nodes

This might seem like a lot of code if you’re attaching lots of similar event listeners to many elements. There are a few ways to go about doing that more efficiently. We learned above that we can get a nodelist of all of the items matching a specific selector with querySelectorAll('selector'). In order to add a listener to each of them we simply need to iterate through the whole list like so:

<div id="container">

<button id="1">Click Me</button>

<button id="2">Click Me</button>

<button id="3">Click Me</button>

</div>

// buttons is a node list. It looks and acts much like an array.

const buttons = document.querySelectorAll('button');

// we use the .forEach method to iterate through each button

buttons.forEach((button) => {

// and for each one we add a 'click' listener

button.addEventListener('click', () => {

alert(button.id);

});

});

This is just the tip of the iceberg when it comes to DOM manipulation and event handling, but it’s enough to get you started with some exercises. In our examples so far we have been using the ‘click’ event exclusively, but there are many more available to you.

Some useful events include:

* click
* dblclick
* keydown
* keyup

You can find a more complete list with explanations of each event on [this page](https://www.w3schools.com/jsref/dom_obj_event.asp).