DOM Manipulation and Events

Mohamed Emary

September 17, 2023

DOM Manipulation and Events

What is DOM, DOM methods and Targeting DOM nodes

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"></` 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

You use CSS "selectors" to target the nodes you want to work with. 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
```

When HTML code is parsed by the browser, it is converted into the DOM. One of the primary differences is that these **nodes are objects** that **have many properties and methods attached to them**. We can use these properties and methods to manipulate the webpage with JS.

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

The return of `querySelectorAll(selectors)` is a **nodelist**, which is similar to an array, but it doesn't have all of the same methods. You can use `Array.from()` to convert it to an array, or use the `...` spread operator to convert it to an array.

```
const divs = document.querySelectorAll('div');
// returns a nodelist of all divs on the page

const divsArray = Array.from(divs);
// converts the nodelist to an array

const divsArray = [...divs];
// converts the nodelist to an array
```

Element creation

To **create an element** use `document.createElement(tagName, [options])`. This creates a new element of tag type `tagName`. `[options]` means you can add some optional parameters to the function. Don't worry about these at this point.

Example: `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:

1. Append elements

`parentNode.appendChild(childNode)` - appends `childNode` as the last child of `parentNode`.

`parentNode.insertBefore(newNode, referenceNode)` - inserts `newNode` into `parentNode` before `referenceNode`.

2. Remove elements

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

Altering elements

If you have a reference to an element you can alter the element's own properties. 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
```

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 dash 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;" // sets the div's background-color by assigning a CSS 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
```

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
```

`textContent` is preferable for adding text, and `innerHTML` should be used sparingly as it can create security risks if misused.

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 file is run whenever 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. To fix this 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: [1]

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

Events

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:

1. You can specify function attributes directly on your HTML elements. *For Example:* `<button onclick="alert('Hello World')">Click Me</button>`
2. You can set properties of form `on[eventType]` (`onclick`, `onmousedown`, etc.) on the DOM nodes in your JavaScript. *For Example:* `<button id="btn">Click Me</button>` on HTML file and in JavaScript file:

   ```
   // the JavaScript file
   const btn = document.querySelector('#btn');
   btn.onclick = () => alert("Hello World");
   ```

3. Or you can attach event listeners to the DOM nodes in your JavaScript. (**The preferred method**) *For Example:* `<button id="btn">Click Me Too</button>` in HTML file and in JavaScript file:

   ```
   // the JavaScript file
   const btn = document.querySelector('#btn');
   ```

[1] Read the second bullet point in this MDN article for more information, which also includes a link to additional script loading strategies.

```
btn.addEventListener('click', () => {
  alert("Hello World");
});
```

Method 1 is not preferred at all as it clutteres the HTML file with JavaScript code.

Both methods 1 and 2 can only attach one event to an element.

Method 3 using `addEventListener` is the preferred way because it's much cleaner and we can attach more than one event to an element

In all 3 methods you can use named functions as well as anonymous functions

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. *For example:*

```
btn.addEventListener('click', function (e) {
  console.log(e);
});
```

`function (e)` is a callback from `addEventListener`. The `e` in that function is an **object that references the event itself**. Within that object you have access to many useful properties and methods (functions that live inside an object) 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';
});
```

If we want to attach the same listener we can this like below:

In HTML:

```
<div id="container">
    <button id="1">Click Me</button>
    <button id="2">Click Me</button>
    <button id="3">Click Me</button>
</div>
```

In JavaScript:

```
// 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 in the
 ↪  node list
buttons.forEach((button) => {

```

```
  // and for each one we add a 'click' listener
  button.addEventListener('click', () => {
    alert(button.id);
  });
});
```

There are many events other than `click` some useful events include:

1. `click`
2. `dblclick`
3. `keydown`
4. `keyup`

Watch Two Videos in *#JavaScript30* Playlist

- Make a JavaScript Drum Kit in Vanilla JS!
- JavaScript Event Capture, Propagation and Bubbling

Watch the two videos especially this video as it contains very valuable information.

References

1. See DOM Enlightenment's section on CSS Style rules for more info on inline styles.
2. See MDN's section on HTML Attributes for more info on available attributes.
3. Example of how `innerHTML` can lead to security risks.
4. Explanation of callbacks can be found HERE.
5. Complete list with explanations of each event on this page.