The DOM

Week 4

A learning experience is one of those things that says, "You know that thing you just did? Don't do that."

- Douglas Adams

Contents

Chapter 1

JavaScript in the Browser

To use JavaScript in the browser we need to use a `<script>` tag. These go at the bottom of the page and come in two forms:

Inline: Can be useful for running tiny bits of code or checking things are working. Generally it's better to use external script files.

```
<script>
    let hello = "Hello, World";
    console.log(hello);
</script>
```

External Script Files: It's best to store our JavaScript in separate `.js` files and then link to them using a `src` attribute:

```
<!-- can be relative, absolute, or remote link -->
<script src="js/app.js"></script>
```

You can't use a `src` tag and inline JS in the same `<script>` tag - only one of them will run.

1.1 JavaScript Lifecycle

When a browser loads a web page it will load the raw HTML from the server and then work its way through each line of code in order[1]. So if you have multiple `<script>` tags on your page they will run in the order that they appear in the HTML.

The JavaScript and any variables that you create using it will only exist until you refresh the page: at that point everything starts again from scratch. *Refreshing the page restarts your JavaScript from scratch*

1.2 IIFEs

You can include as many JavaScript files as you like on a webpage, but you might not have written all of them or be aware of the variables that they've set up. So we need a way to make sure that the variables we declare don't clash with variables that may already exist.

```
// perfectly innocent bit of code
// won't work, as browsers already have a top
// variable defined in global scope
let top = 100;
console.log(top);
```

We can do this by writing a function - which creates new scope - and then call it immediately:

```
let start = () => {
    // no problems as top is scoped to the start function
    let top = 100;
    console.log(top);
};

start();
```

[1]This is a bit of a simplification and not true if the `defer` and `async` attributes are used on a `<script>` tag

However, this still adds a variable called `start` to global scope.

We can tidy it up using an **Immediately Invoked Function Expression**:

```
// create a function
(() => {
    // your code here
    // any variables will be local
})(); // and call it immediately
```

We write an **anonymous function**, wrap it in brackets, and then call it immediately. Because we don't assign the function to a variable we've not added anything to global scope.

It's good practice to wrap all of your code in an IIFE to avoid variable naming issues.

Note: you don't need to do this with ES6 modules (which we'll use later in the course) as they are self-contained by default.

1.3 Developer Tools

Always have the Developer Tools Console open when you're working with JavaScript in the browser. (Mac: `Cmd+Alt+j`/ Windows: `Ctrl+Shift+j`)

You can use the JavaScript console to experiment with different bits of code if you get stuck.

1.3.1 Errors

If there is a single error in your code nothing will work, so make sure you immediately fix any errors that show up in the console.

You could also add the JavaScript Errors Notifier extension to Chrome. This will let you know if there are errors even if you forget to have Developer Tools open.

1.4 Additional Resources

- Eloquent JavaScript: JavaScript and the Browser
- MDN: IIFEs
- IIFEs in Detail
- Deep Dive into the Murky Depths of Script Loading
- Using the Console
- Beyond `console.log()`

Chapter 2

The DOM

The DOM (**Document Object Model**) is how JavaScript represents the HTML on a web page. It turns the elements from the HTML into a JavaScript object that represents the hierarchy of elements.

In all browsers there is a global object called `document` that allows us to interact with the DOM.

```
console.log(document);
```

The `document` object has three important properties that represent the main parts of a page:

```
document.documentElement; // the <html> element
document.head; // the head element
document.body; // the body element
```

These properties represent **HTMLElement** objects, which share various properties and methods that we'll be learning about tomorrow.

2.1 Selecting Individual Elements

The `document` object also has methods that allow us to select elements.

You can use `document.getElementById()` to get an element using its `id`:

```
// get the container element on a page
let container = document.getElementById("container");
```

This returns an `HTMLElement` object representing whichever element in the document has the given ID.[1]

This is the most efficient way to select an element from the DOM - but `id`s need to be unique, so you can't always use this.

You can also use almost any CSS selector to get an element:

```
<html>
  <body>
    <section id="container">
      <ul>
        <li class="menu__item"><a href="/about">About</a></li>
        <li class="menu__item"><a href="/contact">Contact</a></li>
      </ul>
    </section>
  </body>
</html>
```

```
// returns first matching element
let list = document.querySelector("ul") // the ul element

// the first item with menu__item class
let firstMenuItem = document.querySelector(".menu__item")
```

This method also returns an `HTMLElement` object.

[1] If no elements match then you'll get `null` back and none of the element methods will work

2.2 Storing Elements

Once you've selected an element, it's a good idea to store it in a variable so that you can do things with it later:

```
// a variable holding the #container element
let container = document.getElementById("container");
let body = document.body; // a variable holding the <body> element

// later on...
// do some things with container
container.classList.add("current");
```

If you only use an element once in your code then you don't necessarily need to store it in a variable, but it's good to get used to doing it.

IIFEs with Arguments

You've probably noticed that we're using `document` a lot. We can save ourselves a little bit of typing by passing in an argument to our IIFE:

```
// call whatever the first argument is d
(d => {
    // now inside our IIFE we can use d instead of document
    d.getElementById("foo");
})(document); // pass the document object in as the first argument
```

You don't have to call the argument `d`, you could use `doc` or anything else, but `d` is nice and short and fairly obvious. Just don't use a variable named `d` for anything else inside your IIFE.

2.3 Adding and Removing Classes

Once we've got our element, we can start to do things with it. One of the most useful things you can do is add and remove classes in order to change how it appears on screen:

```
let container = document.getElementById("container");

// Add the current class to the #container
container.classList.add("current");

// Remove the hidden class from the #container
container.classList.remove("hidden");
```

2.4 Additional Resources

- Eloquent JavaScript: The DOM
- MDN: `HTMLElement`
- MDN: `classList`
- CSS Selectors
- I Love My IIFE

Chapter 3

Working with Collections

3.1 Selecting Multiple Elements

Sometimes you'll want to work with multiple elements on a page. For example, you might want all the elements that are a particular type or have a particular class.

These methods return an **HTMLCollection**:

```
let menuItems = document.getElementsByClassName("menu__item");
let divs = document.getElementsByTagName("div");
```

These methods return a **NodeList**:

```
// all items with menu__item class
document.querySelectorAll(".menu__item");
// all the <li> elements
document.querySelectorAll("li");
// all the <a> elements inside <li> elements
document.querySelectorAll("li a");
// any elements with a disabled attribute
document.querySelectorAll("[disabled]");
// the first item
document.querySelectorAll("p:first-child");
// any elements with the list or table class
document.querySelectorAll(".list, .table");
```

3.2 NodeLists and HTMLCollections

`NodeList`s and `HTMLCollection`s are **array-like objects**, meaning that they sort of act like arrays, but not really.

Generally, we just want to turn them into an actual array so that they behave how we'd expect. We can do this using `Array.from()`:

```
let items = Array.from(document.getElementsByClassName("menu__item"));
```

Once we've turned them into an array we can use the standard array methods like `forEach()`, `filter()`, `map()`, and `reduce()`.

If you forget to convert them into an array then - as well as missing the array methods that you're used to - there can be some performance issues.

3.3 Using `forEach()`

Once we've converted the selection to an array we can use `forEach()` to work with each item in turn:

```
let items = Array.from(document.getElementsByClassName("menu__item"));

items.forEach(el => el.classList.add("spoon"));
```

We use `forEach()` in this case as many of the DOM methods don't return useful values.

IDs, Classes, and Prefixes

How do you know when to use an `id` or a `class`?

In CSS you should never use `id` selectors, as they make it hard to override rules using cascading. However, in JavaScript it is *much* more efficient to select elements using an `id` than a `class`.

If an element can only ever appear in the HTML once, then use an `id`*. Otherwise, use a* `class`

It's also a good idea to use a special `js__` prefix on any classes you add for use with JavaScript. Although it can look a little verbose, this means that someone editing the styling for an element won't accidentally break your JavaScript.

```
<!-- .list-item is for styling, js__selector is for JavaScript -->
<li class="list-item js__selector">Blah blah</li>
```

You can also add a `js__` prefix to any `id`s - but as `id`s shouldn't be used for styling, this is not strictly necessary.

3.4 Additional Resources

- A Comprehensive Dive into NodeLists
- HTMLCollections and NodeLists
- MDN: `Array.from()`
- MDN: `NodeList`
- MDN: `HTMLCollection`

Chapter 4

Traversing the DOM

4.1 Relationships

The DOM is a **tree** structure, meaning that the elements within it have the following relationships:

- **Parent**: the containing element
- **Child**: the contained element
- **Sibling**: other Child elements with the same Parent

```
<html>
    <body>
        <h1>Lists!</h1>

        <p>A list</p>

        <ul>
            <li>
                <a href="/one">First Thing</a>
            </li>
            <li>
                <a href="/two">Second Thing</a>
            </li>
            <li>
                <a href="/three">Third Thing</a>
            </li>
        </ul>
    </body>
</html>
```

4.2 Traversal

With the DOM you can get elements using their relationships to other elements:

```
let body = document.body; // the <body>

let header = body.firstElementChild; // the first child of body, <h1>
let bodyAgain = header.parentElement; // the <body>

let p = header.nextElementSibling; // the <p>
let ul = p.nextElementSibling; // the <ul>
let headerAgain = p.previousElementSibling; // the <h1>
```

You can also get all the children:

```
let listItems = ul.children;
```

This returns an `HTMLCollection`, so you will want to convert it to an array before you do anything with it.

4.3 Selecting Children

You can use many of the selection methods on any element:

```
container.getElementsByClassName("menu__item");
container.getElementsByTagName("li");
container.querySelector("li a");
container.querySelectorAll("li a");
```

It can be much more efficient to select a container element (particularly if you use `getElementById`) and then use these methods to select child elements.

It's also preferable to use the `getElement...` methods over the `querySelector...` methods where you can as they are generally much faster.

4.4 Additional Resources

- How to traverse the DOM

Chapter 5

Manipulating the DOM

5.1 Querying Elements

You can find out all sorts of information about an element:

```
let container = document.getElementById("container");

// the height of the element (in pixels)
container.clientHeight;
// the width of the element (in pixels)
container.clientWidth;
// the inline style border colour
container.style.borderColor;
// the border colour - including CSS
window.getComputedStyle(container).borderColor;
// the html inside the element (e.g. "<p>Text</p>")
container.innerHTML;
// the text inside the element (e.g. "Text")
container.textContent;
// returns the position of the element relative to the page
container.getBoundingClientRect();
```

You can store these in variables to use later:

```
let height = container.clientHeight;
// add 300 pixels to the height of the container
container.style.height = (height + 300) + "px";
```

It's always better to not query the DOM if you can avoid it. If you can use variables to keep track of changes your code will be much easier to understand and it will run faster.

5.2 `document` and `window` Properties

You can find useful information out about the page[1]:

```
window.pageYOffset; // the current vertical scroll position
window.pageXOffset; // the current horizontal scroll position

window.innerHeight; // the height of the viewport
window.innerWidth; // the width of the viewport

document.body.clientHeight; // the height of the document
document.body.clientWidth; // the width of the document
```

5.3 Manipulating Elements

You can also edit the styling of the element directly by setting sub-properties of the `style` property[2]:

```
let container = document.getElementById("container");

container.style.border = "1px solid red";
container.style.position = "absolute";
container.style.left = "20px";
container.style.top = "20px";

// notice we have to write marginTop, not margin-top
// due to property naming rules
container.style.marginTop = "20px";
```

[1] The `window` object is part of the **BOM** (Browser Object Model). As well as having the window specific properties/methods, it actually represents the **global scope** in a browser.
[2] This is part of the **CSSOM** (CSS Object Model) - we do like our object models in JavaScript

You can set the height and width too:

```
let container = document.getElementById("container");

// make sure you include the units (px in this case)
container.style.height = "200px";
container.style.width = "200px";
```

It is preferable to use CSS classes where you can, but sometimes you will need to use the `.style` property if the styling is dependent on values calculated by JavaScript.

You can also change the text inside an element:

```
let title = document.getElementById("title");
title.textContent = "New Title"; // replaces text of #title
```

Be careful, *setting `.textContent` will remove everything inside the element.*

5.4 Attributes

We can query/edit any attribute of an element:

```
<input id="age" name="age" value="20" />
```

```
let input = document.getElementById("age");

input.getAttribute("name"); // returns "age"
input.getAttribute("value"); // returns "20" (as a string)
input.setAttribute("value", "600"); // set the value attribute to "600"
input.removeAttribute("disabled"); // remove the disabled attribute
```

5.5 Form Fields

To get the *actual* value of the input (as opposed to the default value), we can use the `.value` property:

```
input.value; // a string containing the current value of the input
```

The `.value` property will *always* return a string - so be careful if you're doing any addition!

We can also call the `.focus()` method on a form field to give it focus:

```
input.focus(); // gives the input focus
```

5.6 Data Attributes

Sometimes we want to let JavaScript know something about an element that isn't a standard property. We can use `data-*` attributes for this.

For example, if we wanted to store additional information about some books:

```
<ul id="books">
    <li data-id="12" data-author="Marijn Haverbeke">
        Eloquent JavaScript
    </li>
    <li data-id="35" data-author="Douglas Crockford">
        JavaScript: The Good Parts
    </li>
    <li data-id="59" data-author="David Flanagan">
        JavaScript: The Definitive Guide
    </li>
</ul>
```

We could then access this using the `dataset` property:

```
let first = document.getElementById("books").firstElementChild;

if (first) {
    console.log(first.dataset.id);
    console.log(first.dataset.author);
}
```

You should only use `data-*` attributes as a last resort, as reading from the DOM is slow. Ideally the data would be used and stored only in JavaScript.

5.7 Additional Resources

- MDN: `window.getComputedStyle()`
- MDN: `getBoundingClientRect()`
- MDN: `data-*` Attributes
- MDN: `window`

Chapter 6

Creating, Moving & Removing Elements

6.1 Creating Elements

You can create elements and add them to the page using `appendChild()`:[1]

```
// create a new <p> element and store it in a variable
let p = document.createElement("p");

// set the text inside the <p> to "A paragraph"
p.textContent = "A paragraph";

let container = document.getElementById("container");
container.appendChild(p); // append the new <p> to the container
```

Don't use `innerHTML` *to add elements!*

6.2 Document Fragments

Every time you add an element to the DOM the browser will redraw the webpage. This can be quite a slow process. If we're only adding a few items to the DOM then it probably won't have a huge affect on performance, but if we were adding 1,000 items it could be noticeably slower.

[1] You can also use `insertBefore()` to add an element to the DOM - we'll cover this in a bit

Rather than appending items to the DOM one by one, we can use a **Document Fragment** to prepare them all, and then add them in one go.

```
// the list is already on the page, so if we append
// to it we will cause a redraw
let list = document.getElementById("list");
let fragment = document.createDocumentFragment();

for (let i = 0; i < 1000; i += 1) {
    let el = document.createElement("li");
    el.textContent = "Blah";

    // append to the document fragment
    // this isn't on the page, so won't cause a redraw
    fragment.appendChild(el);
}

// append the fragment once
// this appends all 1000 items in one go
// so we only get one redraw instead of 1000
list.appendChild(fragment);
```

6.3 Inserting/Moving Elements

Sometimes we want to move the position of elements or insert elements that are not currently on the page.

We can use the `appendChild()` and `insertBefore()` methods to do this:

```
// create a new element
let p = document.createElement("p");
p.textContent = "Hello, world";

// select an existing element
let container = document.getElementById("container");

// puts the paragraph inside the container as the last element
container.appendChild(p);
// puts the paragraph inside the container as the first element
container.insertBefore(p, container.firstElementChild);
```

If the element already exists on the page, running these commands will move the element around the page.

We can also use the `insertAdjacentElement` method to place elements.

```
// add the element before the target
target.insertAdjacentElement("beforebegin", element);

// add the element after the target
target.insertAdjacentElement("afterend", element);
```

The first argument can take one of four strings:

1. `"beforebegin"`: before the element itself
2. `"afterbegin"`: just inside the element, before its first child
3. `"beforeend"`: just inside the element, after its last child
4. `"afterend"`: after the element itself

6.4 Removing Elements

You can remove elements from a page use the `.remove()` method:

```
// find the element with id mobile-nav
let mobileNav = document.getElementById("mobile-nav");

// remove the element from the page
mobileNav.remove();

// we can reinsert the item later
document.body.insertBefore(mobileNav, document.body.firstElementChild);
```

jQuery

jQuery is JavaScript library that makes working with the DOM much nicer - it can also support browsers going back to IE8.

```
// selecting
// native DOM
let items = Array.from(document.querySelectorAll(".menu_item"));
// jQuery
let items = $(".menu_item");

// prepending
// native DOM
body.insertBefore(mobileNav, body.firstElementChild);
// jQuery
body.prepend(mobileNav);

// setting text and adding a class in native DOM
body.textContent = "Blah";
body.classList.add("foo");

// setting text and adding a class in jQuery
body.text("Blah").addClass("foo");
```

Over the years browsers have slowly adopted many of the jQuery methods and concepts like `remove` and `querySelector`.

If you're working on a site that already uses jQuery you'd be mad not to use it. However, jQuery is an additional download, so you shouldn't add it to sites unless you're using it a lot.

6.5 Additional Resources

- MDN: Document Fragments
- The problem with innerHTML
- jQuery: Getting Started

Chapter 7

Events

7.1 Event Driven Programming

Almost everything you do in JavaScript will be a response to an event.

Some examples:

- the page loading
- user clicking an element
- user submitting a form
- user moving the mouse
- resizing the window

Events allow us to respond to a user's actions.

We use the `addEventListener` method to tell the browser what we want to happen when the event is **triggered**.

We pass `addEventListener` a function, which gets called by the browser each time the registered event occurs. We call such a function an **event handler**:

```
// this runs straight away
console.log("page loaded");

let container = document.getElementById("container");

// this runs when the element is clicked
container.addEventListener("click", () => console.log("clicked"));

// this runs when the mouse moves over the element
container.addEventListener("mousemove", () => console.log("mouse moving"));
```

There are all sorts of events:

- `keydown`: when a key is pressed down
- `keyup`: when the key is released
- `click`: when the element is clicked
- `mousedown`: when the mouse is pressed down
- `mouseup`: when it comes back up again
- `focus`/`blur`: when a form field gets/loses focus
- `change`/`input`: fires when a form input changes
- `submit`: fired on submitting a form

A full list is available on the MDN site

Some events will only apply to specific types of elements (e.g. you can only submit a form)

7.2 Window Events

Some events need to be on the `window`: most usefully, resizing and scrolling the page.

```
window.addEventListener("scroll", () => {
    console.log("scrolling");
});

window.addEventListener("resize", () => {
    console.log("resizing");
});
```

7.3 State

Event handlers are **short-lived**: they are triggered by an event, run the code inside, and then they're done. Any variables that are declared *inside* an event handler will only exist temporarily.

Your main application code is (comparatively) **long-lived**: any variables will exist as long as the page isn't refreshed.

This means if we want to keep track of any values *between* events, we need to make sure the variables live *outside* our event handlers. This is what we refer to as **state**: variables we use to keep track of changes in our app.

```
// long-lived variables
let increment = document.getElementById("increment");

// the state
// keep track of a value that changes over time
let counter = 0;

// event handlers need to refer to variables outside their local scope
// otherwise they can't keep track of anything
increment.addEventListener("click", () => counter += 1);
```

Remember, *you should avoid querying the DOM if you can*. Keep your state in JavaScript and use variables to keep track of changes.

7.4 Additional Resources

- MDN: `addEventListener`
- State
- Eloquent JavaScript: Events

Chapter 8

Advanced Event Handling

8.1 The Event Object

Whenever you set up an event handler the first argument passed to your function is the **event object**.

```
let input = document.getElementById("input");

input.addEventListener("keyup", event => {
    if (event.key === "Escape") {
        input.value = "";
    }
});
```

The event object has all sorts of useful properties:

- Keyboard events: The key that was pressed (`event.key`)
- Mouse events: The coordinates of the mouse (`event.clientX`, `event.clientY`)
- The event type, e.g. `mousedown` (`event.type`)

8.2 .preventDefault()

By default the browser will run the function and then resume its normal behaviour.

For example, if you registered a `click` event on a link, the browser would run your code, but then follow the link, which would load a new page. You can prevent this using `event.preventDefault()`:

```
let link = document.getElementById("link");

link.addEventListener("click", event => {
    event.preventDefault();

    // do something...
});
```

This will be necessary for any event that navigates away from the current page, e.g. submitting a form, clicking a link

8.3 Bubbling

Events **bubble** up the page hierarchy: all parent elements of the element that the event was fired on will also fire the event.

```
<div id="container">
    <ul>
        <li>
            <a href="/link">Link</a>
        </li>
    </ul>
</div>
```

We can use `event.target` to find out which element the event originated from.

```
let container = document.getElementById("container");

container.addEventListener("click", event => {
    let clicked = event.target;
    clicked.classList.add("clicked");
});
```

8.3.1 Event Delegation

This allows to add an event for multiple items in an efficient manner:

```
<ul id="list">
    <li><a href="/one">One</a></li>
    <li><a href="/two">Two</a></li>
    ...
    <li><a href="/one-thousand">One Thousand</a></li>
</ul>
```

We can listen on the parent `<ul>` rather than setting up an event handler for every list item:

```
let list = document.getElementById("list");

list.addEventListener("click", e => {
    let clicked = e.target;

    // only add class if it's a link
    if (clicked.tagName === "A") {
        e.preventDefault();
        clicked.classList.add("clicked");
    }
});
```

`.matches()`

Sometimes when we're using event delegation our conditional might get quite complicated. If that's the case we can use the `.matches()` method to check if the element matches a given CSS selector.

```
<ul id="list">
    <li class="list-item">
        <a href="/one">One</a>
    </li>
    <li class="list-item">
        <a href="/two">Two</a>
    </li>
    ...
    <li class="list-item current">
        <a href="/one-thousand">One Thousand</a>
    </li>
</ul>
```

```
let list = document.getElementById("list");

list.addEventListener("click", e => {
    let clicked = e.target;

    // only add class if it matches the given CSS selector
    if (clicked.matches("li.list-item.current a")) {
        e.preventDefault();
        clicked.classList.add("clicked");
    }
});
```

8.4 Additional Resources

- How JavaScript Event Delegation Works
- MDN: `.matches()`
- Debouncing & Throttling

Colophon

Created using TEX

Fonts

Feijoa by Klim Type Foundry
Gill Sans by Eric Gill
Fira Mono by Carrois Apostrophe

Colour Palette

Solarized by Ethan Shoonover

Written by Mark Wales
smallhadroncollider.com
January 25, 2019