4

JavaScript in the Browser

with the DOM

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

- Douglas Adams

Contents

How To Use This Document

Bits of text in magenta are links and should be clicked at every opportunity. Bits of text in `monospaced yellow` represent code. Most the other text is just text: you should probably read it.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a [View on GitHub] link underneath them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I've switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn't need to take any additional notes. However, I know that this doesn't work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into "smart-quotes". These can be almost impossible to spot in a text-editor, but will completely break your code.

Chapter 1

JavaScript in the Browser

So far we've been doing all of our JavaScript on the command-line using Node. This has allowed us to focus on learning the fundamentals of programming without any distractions.

When moving into the browser, the key thing to remember is that JavaScript in the browser is *still* JavaScript: we'll be learning about ways to manipulate HTML elements and deal with user interaction, but we'll be doing it using functions and variables and strings and objects and conditionals and loops and all the other stuff we learnt about last week. It's a new environment, but don't forget the skills you've learnt so far.

So let's take everything we know and make our HTML dance! Or at the very least turn the border of a `<div>` red. Truly we live in interesting times.

1.1 `<script>`

To use JavaScript in the browser we need to use the `<script>` tag.

1.1.1 External Scripts

It's best to store our JavaScript in separate `.js` files and then link to them using a `src` attribute. This way you can use the same JavaScript file in multiple HTML files.

The script tag should go in the `<head>` section of your HTML file and should have a `defer` attribute. If you forget the `defer` attribute things will not work as you expect.

```html
<script defer src="js/app.js"></script>
```

The `src` attribute can be a relative, absolute, or remote link:

```html
<!-- relative -->
<script defer src="js/app.js"></script>

<!-- absolute -->
<script defer src="/resources/js/app.js"></script>

<!-- remote -->
<script defer src="https://cdn.wombat.com/app.js"></script>
```

When you include a `.js` file for the first time it can be a good idea to just shove `console.log("Hello!")` in it, then open up the Developer Tools console and check that it's definitely working.

The Wandering `<script>`

`<script>` is a bit of a nomad. When JS was first added to the browser we put `<script>` tags in the `<head>` (without a `defer` attribute). When JS files started to get quite big in the early naughties this started to cause issues, as the browser wouldn't render anything until the script had fully downloaded, so pages seemed unresponsive.

So `<script>` began its migration to the bottom of the `<body>` and that's where it lived for much of the last 15 years. It was never meant to live inside the body of HTML (that's where the stuff a user sees lives), but if you put it as the last thing on the page everything else renders before the JS starts loading, so the page feels more responsive.

Then `defer` came along and `<script>` started to migrate back to the `<head>`. The `defer` attribute just tells the browser to download the JS in the background, but not run it *until* the rest of the page has finished loading and rendering. So it lets us keep `<script>` in the `<head>` (where it belongs), without the associated performance issues.

This is an ongoing migration, so it's not uncommon to still see the `<script>` tag at the bottom of the `<body>`. If you find a `<script>` tag still there, it's probably best to leave it where it is so as not to startle it.

1.1.2 Inline Scripts

It can sometimes be useful to run tiny bits of code to check things are working without creating a separate `.js` file.

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

However, it's better to use external script files if you can.

∞

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

1.2 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**

1.3 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 that won't work
// 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();
```

However, this still adds a variable called `start` to global scope, which could still cause issues.

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

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 a single IIFE to avoid variable naming issues.[2]

1.4 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.

1.4.1 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.

[2] 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.

1.5 Additional Resources

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

Chapter 2

The DOM

We need a way to use JavaScript to communicate with HTML elements on a page. The browser provides an interface known as 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. By reading and setting these properties we can make changes to elements on the page.

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 Getting Individual Elements

The `document` object also has methods that allow us to get HTML elements to work with.

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 get 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 got an element from the DOM, 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 parameter `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");
```

The `classList` methods don't return anything when you call them. This means you can't chain them or do anything useful with the value you get back (it will always be `undefined`).

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

When writing HTML to be used with JavaScript, 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

- 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
<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>
```

In the above example the `<body>` is the parent of the `<h1>`, the `<p>`, and the `<ul>`. The `<ul>` is the parent of the three `<li>`s. The `<li>`s are all children of the `<ul>` and are siblings with each other. Each `<a>` is the child of an `<li>`, but have no children[1] or siblings themselves.

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");
```

[1]This is not technically true as the `<a>`'s each contain text, which in the DOM is represented as a `TextNode`, but it is rare to use these in modern JavaScript

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 height of the element, including borders (in pixels)
container.offsetHeight;
// the width of the element, including borders (in pixels)
container.offsetWidth;
// returns the position of the element relative to the page
container.getBoundingClientRect();
// the text inside the element
container.textContent;
// the inline style border colour
container.style.borderColor;
```

You can store these values 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.offsetHeight; // the height of the document
document.body.offsetWidth; // 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";

// move element around the page
container.style.transform = "translateX(20px)";
container.style.transform = "translateY(20px)";
container.style.transform = "translate(20px, 20px)"; // (X, Y)

// 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`
- MDN: Determining the dimensions of elements

Chapter 6

Creating, Moving & Removing Elements

6.1 Creating Elements

You can create elements using `document.createElement()`:

```
// 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";
```

This creates an `HTMLElement` object instance, just as if you'd got it from the DOM. However, it only exists in JavaScript land until you add it to the DOM.

You can add an element to the DOM using `.append()`:

```
// select an item from the DOM
let container = document.getElementById("container");
// append the new <p> to the container element
// this adds it to the DOM
container.append(p);
```

Notice that we set the `.textContent` property of the `<p>` *before* we add it to the DOM. If we added it to the DOM and then changed the property the browser would need to re-render the page twice. If we make the change before we add it to the

DOM, the browser only re-renders once. Re-rendering is very costly in terms of processing power, so it's best to avoid doing it unnecessarily.

There are four useful methods for adding elements to the DOM:[1]

- `el.prepend(otherEl)`: adds `otherEl` as the first element of `el`
- `el.append(otherEl)`: adds `otherEl` as the last element of `el`
- `el.before(otherEl)`: adds `otherEl` before `el`
- `el.after(otherEl)`: adds `otherEl` after `el`

Polyfills

The methods listed above are part of the "DOM Level 4" spec, which is all rather new. Until quite recently doing the equivalent of these methods took rather more code (and bizarre code at that). Because they're new, they're not supported in older browsers, which might seem like a bit of a non-starter.

Luckily, it's possible to "monkey patch" JavaScript: adding functionality to JavaScript using JavaScript itself. Generally, this is not to be encouraged, as it means your version of JavaScript is different from the standard, which could be confusing for other consumers of your code. But it can be safely used to make older JavaScript engines support new functionality. We call such code a **polyfill**.[2]

The DOM4 Polyfill can be added to the `<head>` of a page to make sure that the above methods are all present.

[1]There's also `.appendChild()`, `.insertBefore()`, and the bizarre `.insertAdjacentElement()`, but the newer methods listed above are much nicer to use
[2]Which are a type of **shim**: code that intercepts other code

6.2 Inserting/Moving Elements

Sometimes we want to move the position of elements on the page.

We can use `.append()` and friends 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
// moves it rather than copying it
container.append(p);
```

Remember, the DOM models each element as an object, and objects are stored by reference. So if we try and `.append()` an object already in the DOM, it moves it rather than copying it. This is also true of `.prepend()`, `.before()`, and `.after()`.

6.3 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.prepend(mobileNav);
```

6.4 Document Fragments

Every time you add an element to the DOM the browser will re-render the webpage. This can be quite a slow process.

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 animals = ["wombat", "kangaroo", "wombat", "platypus", "koala"];
let fragment = document.createDocumentFragment();

animals.forEach(animal => {
    let el = document.createElement("li");
    el.textContent = animal;

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

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

The document fragment creates a sort of phantom element, that you can add items to as if it were real, but when you add it to the DOM it’s as if it never existed.

You only need to use a document fragment if the item you want to add multiple items to is already on the page. If the parent element hasn’t yet been added to the page, then adding items to it won’t cause a re-render, so document fragments aren’t necessary.

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");

// 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()`, `.before()` and `document.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
- You Might Not Need jQuery
- 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
- Pop-Up Trombone: the best use of the `resize` event ever

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 and methods:

- The key that was pressed (`event.key`)
- The coordinates of the mouse or finger(s) (`event.clientX`, `event.clientY`)
- The time the event occurred
- The type of the event

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 fire on an element whether you're listening to them or not.

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>
```

If we clicked on the `<a>` then it would fire a `click` even on the `<a>`, but then it would also fire one on the `<li>`, then the `<ul>`, then the `<div>`, all the way up to the `document` and then `window` objects.

8.3.1 `event.target`

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");
});
```

8.3.2 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");
    }
});
```

Delegation has the added advantage that it will work for *new* HTML elements that are added after the event listener is registered. That's because we're not listening on the new item itself, just on the parent that contains it.

`.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.

```html
<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>
```

```javascript
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");
    }
});
```

8.3.3 `event.composedPath()`

We can also use the `event.composedPath()` property to see the full route of an event:

```
let cards = d.getElementById("cards");

cards.addEventListener("click", e => {
    // get the path of the bubble event
    let path = e.composedPath(); // in older browsers: e.path ||
    ↪  e.composedPath()

    // go over each item in the event path
    // check if it has a .matches() method (sometimes it's not there)
    // if it does then see if the element matches some CSS selector
    if (path.some(el => el.matches ? el.matches(".card") : false)) {
        console.log("card clicked");
    }
});
```

8.4 Removing Event Listeners

Sometimes it's useful to remove an event listener from an element. In order to do this we need to have stored the event listener function in a variable, otherwise we can't tell the DOM *which* event listener we're trying to remove:[1]

```
let container = document.getElementById("container");
let count = 0;

let clickHandler = () => {
    count += 1;
    container.textContent = count;

    if (count > 10) {
        // if count is clicked more than 10 times
        // remove the event listener
        container.removeEventListener("click", clickHandler);
    }
```

[1] Although we've not seen an example of this, it's possible to add as many event listeners as you like for the same event. This can be handy to avoid writing functions that try to do too many things.

```
};

// add the event listener using the named function
container.addEventListener("click", clickHandler);
```

Event listeners use memory, so it's important to remove them if you don't need them any more. It's very common in JavaScript apps to see memory usage increase over time. This is usually because the code removes elements from the DOM but doesn't remove the event listeners - the DOM won't do this for you, as you may reinsert the element at a later point. Libraries like React, which we'll cover later in the course, do this automatically.

8.5 Additional Resources

- How JavaScript Event Delegation Works
- MDN: `.matches()`
- MDN: `.removeEventListener()`
- Debouncing & Throttling
- 8 DOM Features You Didn't Know Existed

Glossary

- **Bubbling**: events handlers fire on the element on which they occur, and then work their way up the DOM and fire on every parent element
- **Delegation**: adding an event listener higher up the DOM and taking advantage of bubbling for more efficient event handling on multiple elements
- **Document Object Model (DOM)**: the interface browsers provide to interact with HTML elements from JavaScript
- **Monkey Patching**: using a programming language to add features to itself
- **Polyfill**: a piece of code that retrofits older browsers with newer features using monkey-patching. A type of shim
- **Shim**: a piece of code that intercepts function calls, allowing the addition of backwards compatibility
- **State**: long-lived variables used to keep track of changes over time
- **Tree**: a data structure which can be expressed using the **parent**, **child**, and **sibling** relationships

Colophon

Created using $\TeX$

Fonts

- Feijoa by Klim Type Foundry
- Avenir Next by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- Fira Mono by Carrois Apostrophe

Colour Palette

- Solarized by Ethan Shoonover

Written by Mark Wales

∞

August 14, 2019