Lesson Tuesday

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / Further Exploration: IIFEs

Text

This lesson is **optional further exploration**. You can skip it entirely if you want. We'll use a more advanced technique called an IIFE to refactor the code we wrote in the last lesson.

Immediately-Invoked Function Expressions

An IIFE is an Immediately-Invoked Function Expression. IIFEs definitely have their use cases (there's more on this below) but they really aren't necessary for beginners or even intermediate developers to know about. However, you may run into them in the wild, so it can be helpful to get some exposure now.

So, to get familiar with how we can use an IIFE, let's turn our async `getWeather` function into an IIFE that's a part of our `handleFormSubmission`. This is what our `index.js` file will look like:

src/index.js

```
import 'bootstrap';
import 'bootstrap/dist/css/bootstrap.min.css';
import './css/styles.css';
import WeatherService from './weather-service.js';

// UI Logic

function handleFormSubmission(event) {
  event.preventDefault();
  const city = document.querySelector('#location').value;
  document.querySelector('#location').value = null;
  (async function() {
    const response = await WeatherService.getWeather(city);
    if (response.main) {
      printElements(response, city);
    } else {
      printError(response, city);
    }
  })();
}

function printElements(response, city) {
  document.querySelector('#showResponse').innerText = `The
humidity in ${city} is ${response.main.humidity}%.
  The temperature in Kelvins is ${response.main.temp} degre
es.`;
}

function printError(error, city) {
  document.querySelector('#showResponse').innerText = `Ther
e was an error accessing the weather data for ${city}:
  ${error}.`;
}

window.addEventListener("load", function() {
  document.querySelector('form').addEventListener("submit",
handleFormSubmission);
});
```

As noted, we've removed the `getWeather` function, and now we have an IIFE inside of `handleFormSubmission` that does the same thing as the `getWeather` function:

```
function handleFormSubmission() {
  ...

  (async function() {
    const response = await WeatherService.getWeather(city);
    if (response.main) {
      printElements(response, city);
    } else {
      printError(response, city);
    }
  })();
}
```

IFFE Syntax

So what exactly is going on here? And what's with the weird additional parens? Well, that's the syntax for an IIFE — a function that's immediately invoked. Here's the syntax in a nutshell:

```
// Pseudocode example!

(function() {
  // A function to be invoked immediately.
})();
```

The key thing here is that we wrap the function itself in grouping parentheses (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Grouping) and then we also have a closing parens at the end of the IIFE:

- The parentheses around the function are so the JavaScript compiler doesn't throw an error.

- The parens at the end invoke the function just like we'd invoke any named function.

We can even use a named function for an IIFE:

```
// Pseudocode!

(function thisIsImmediatelyInvoked() {
  // A function to be invoked immediately.
})();
```

Benefits of IIFEs

Updating our code in `index.js` to use an IIFE changed the way our code was structured, but it wasn't necessarily an improvement. So, the question remains, why use an IIFE?

Well, the biggest benefit of IIFEs is data privacy. Any variables or data in an IIFE are scoped to it and aren't available elsewhere. If we don't use an IIFE, those variables will be available in other scopes, perhaps even the global scope, which isn't good.

Let's quickly demonstrate:

```
> (function() {
  const secret = "This is a secret!"
})();
> secret;
VM126:4 Uncaught ReferenceError: secret is not defined
```

If we run this in the DevTools console, we'll get a `secret is not defined` error. As we know, it's very important to scope our code. For instance, think about the projects we did before we implemented JavaScript. We might have something like this:

```
function doSomething() {
  // This function will do something.
}

window.onload = function() {
  doSomething();
};
```

Well, that `doSomething()` function is available globally — we could even access it by typing in `window.doSomething` in the DevTools console when we run our project.

So traditionally, it was common to use an IIFE to wrap the code in the user interface source code like this:

```
(function() {

  function doSomething() {
    // This function will do something.
  }

  window.onload = function() {
    doSomething();
  };
})();
```

This way, the `doSomething()` function will no longer be in the global scope (or accessible via the `window` object). It would just be available within the scope of the IIFE, which is where we want it.

Fortunately, webpack solves this problem for us, too, and doesn't allow code to leak into the global scope, so you won't see an IIFE used in this way with webpack. They can still be a useful tool for scoping code, though, and they can make an async function look a little bit cleaner.

The example repository hasn't been updated to include this code because it's optional. However, if you are interested in working with IIFEs, we recommend trying them out when you are writing async functions. For more information on the use cases for IIFEs, visit the MDN documentation on IIFEs (https://developer.mozilla.org/en-US/docs/Glossary/IIFE).

Previous (/intermediate-javascript/asynchrony-and-apis/async-and-await)
Next (/intermediate-javascript/asynchrony-and-apis/bike-index-cryptocurrency-analytics-app-api-of-choice-two-day-project-part-1)
Lesson 26 of 33

Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)