Running Scripts In The Browser

Timing is everything, in life as well as in code that runs in a browser. Executing a script at the right time is an important part of developing front-end code. A script that runs too early or too late can cause bugs and dramatically affect user experience. After reading this section, you should be able to utilize the proper methods for ensuring your scripts run at the right time.

In previous sections, we reviewed how the DOM and BOM works and used event listeners to trigger script execution. In this lesson, we'll dig deeper into the `window` object and learn multiple ways to ensure a script runs after the necessary objects are loaded.

Using the Window API

The `window` object, the core of the Browser Object Model (BOM), has a number of properties and methods that we can use to reference the window object. Refer to the MDN documentation on the Window API for a detailed list of methods and properties. We'll explore some of these methods now to give you a better grasp on what the `window` object can do for you.

Let's use a Window API method called `resizeTo()` to change the width and height of a user's window in a script.

```
// windowTest.js

// Open a new window
newWindow = window.open("", "", "width=100, height=100");

// Resize the new window
newWindow.resizeTo(500, 500);
```

You can execute the code above in your web browser in Google Chrome by right clicking the page, selecting inspect, and selecting the console tab. Paste the code above into the console. When you do this, make sure you are not in full-screen mode for Chrome, otherwise you won't be able to resize the new window!

Note: You must open a new window using `window.open` before it can be resized. This method won't work in an already open window or in a new tab.

Check out the documentation on Window.resizeTo() and Window.resizeBy() for more information.

Go to wikipedia and try setting the window scroll position by pasting `window.scroll(0,300)` in the developer console (right click, inspect, console like usual). Play around with different scroll values. Pretty neat, huh?

Context, scope, and anonymous functions

Two important terms to understand when you're developing in Javascript are **context** and **scope**. Ryan Morr has a great write-up about the differences between the two here: "Understanding Scope and Context in Javascript".

The important things to note about **context** are:

1. Every function has a context (as well as a scope).
2. Context refers to the object that *owns* the function (i.e. the value of *this* inside a given function).
3. Context is most often determined by how a function is invoked.

Take, for example, the following code:

```
const foo = {
  bar: function() {
```

```
    return this;
  }
};
console.log(foo.bar() === foo);
// returns true
```

The anonymous function above is a method of the `foo` object, which means that `this` returns the object itself — the context, in this case.

What about functions that are unbound to an object, or not scoped inside of another function? Try running this anonymous function, and see what happens.

```
(function() {
  console.log(this);
})();
```

When you open your console in the browser and run this code, you should see the `window` object printed. When a function is called in the global scope, `this` defaults to the global context, or in the case of running code in the browser, the `window` object.

Refer to "Understanding Scope and Context in Javascript" for more about the scope chain, closures, and using `.call()` and `.apply()` on functions.

Running a script on DOMContentLoaded

Now you will learn how to run a script on `DOMContentLoaded`, when the document has been loaded without waiting for stylesheets, images and subframes to load.

Let's practice. Set up an HTML file, import an external JS file, and run a script on `DOMContentLoaded`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="dom-ready-script.js"></script>
  </head>
  <body></body>
  <html></html>
</html>
```

JS

```
window.addEventListener("DOMContentLoaded", event => {
  console.log("This script loaded when the DOM was ready.");
});
```

Running a script on page load

`DOMContentLoaded` ensures that a script will run when the document has been loaded without waiting for stylesheets, images and subframes to load. However, if we wanted to wait for **everything** in the document to load before running the script, we could instead use the `window` object method `window.onload`.

Let's practice it here. Set up an HTML file, import an external JS file, and run a script on `window.onload`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="window-load-script.js"></script>
  </head>
  <body></body>
  <html></html>
</html>
```

JS

```
window.onload = () => {
  console.log(
    "This script loaded when all the resources and the DOM were ready."
  );
};
```

Ways to prevent a script from running until page loads

There are actually multiple ways to prevent a script from running until the page has loaded. We'll review three of them here:

1. Use the `DOMContentLoaded` event in an external JS file
2. Put a script tag importing your external code at the bottom of your HTML file
3. Add an attribute to the script tag, like `async` or `defer`

We've reviewed the first method above. Let's now review numbers **2** and **3**. If you want to make sure that all your HTML has loaded before a script runs, an easy option is to include your script immediately after the HTML you need. This works because HTML builds the DOM tree in the order of how your HTML file is structured. Whatever is on top will be loaded first, such as script tags in the `<head>`. It makes sense, then, to keep your script at the bottom of your HTML, right before the closing `</body>` tag, like below.

```
<html>
  <head></head>
  <body>
    …
    <script src="script.js"></script>
  </body>
</html>
```

If you want to include your script in the `<head>` tags, rather than the `<body>` tags, there is another option: We could use the `async` or `defer` methods in our `<script>` tag. Flavio Copes has a great write-up on using `async` or `defer` with graphics showing exactly when the browser parses HTML, fetches the script, and executes the script.

With `async`, a script is fetched asynchronously. After the script is fetched, HTML parsing is paused to execute the script, and then it's resumed. With `defer`, a script is fetched asynchronously and is executed only after HTML parsing is finished.

You can use the `async` and `defer` methods independently or simultaneously. Newer browsers recognize `async`, while older ones recognize `defer`. If you use `async defer` simultaneously, `async` takes precedence, while older browsers that don't recognize it will default to `defer`. Check `caniuse.com` to see which browsers are compatible with async and defer.

```
  <script async src="scriptA.js"></script>

  <script defer src="scriptB.js"></script>

  <script async defer src="scriptC.js"></script>
```

What we learned:

- How to use Window API methods
- The context and scope of a function
- Review of `DOMContentLoaded` and `window.onload`
- How to prevent a script from running until a page loads