Browser Basics Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Explain the difference between the BOM (browser object model) and the DOM(document object model).
2. Given a diagram of all the different parts of the Browser identify each part. Use the Window API to change the innerHeight of a user's window.
3. Identify the context of an anonymous functions running in the Browser (the window).
4. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when all the elements on the page load (using`DOMContentLoaded`)
5. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when the page loads
6. Identify three ways to prevent JS code from executing until an entire HTML page is loaded
7. Label a diagram on the Request/Response cycle.
8. Explain the Browser's main role in the request/response cycle. (1.Parsing HTML,CSS, JS 2. Rendering that information to the user by constructing a DOM tree and rendering it)
9. Given several detractors - identify which real-world situations could be implemented with the Web Storage API (shopping cart, forms saving inputs etc.)
10. Given a website to visit that depends on cookies (like Amazon), students should be able to go to that site add something to their cart and then delete that cookie using the Chrome Developer tools in order to empty their cart.

Browsers: They're The BOM Dot Com!

If the Internet exists, but there's no way to browse it, does it even really exist? Unless you've been living under a rock for the past couple decades, you should know what a browser is, and you probably use it multiple times a day. Browsers are something most people take for granted, but behind the scenes is a complex structure working to display information to users who browse the Web.

Web developers rely on browsers constantly. They can be your best friend or your worst enemy. (*Yes, we're looking at you, IE!*) Spending some time learning about browsers will help you get a higher-level understanding of how the Web operates, how to debug, and how to write code that works across browsers. In this reading, we'll learn about the BOM (Browser Object Model), how it's structured, and how it differs from the DOM (Document Object Model).

The DOM vs. the BOM

By now, you've learned about the **DOM, or Document Object Model**, and that it contains a collection of nodes (HTML elements), that can be accessed and manipulated. In essence, the `document` object is a Web page, and the DOM represents the object hierarchy of that document.

How do we access a document on the Web? Through a browser, of course! If we took a bird's-eye view of the browser, we would see that the document object is part of a hierarchy of browser objects. This hierarchy is known as the **BOM, or Browser Object Model**.

The chief browser object is the `window` object, which contains properties and methods we can use to access different objects within the window. These include:

- `window.navigator`
  - Returns a reference to the navigator object.
- `window.screen`
  - Returns a reference to the screen object associated with the window.
- `window.history`
  - Returns a reference to the history object.
- `window.location`
  - Gets/sets the location, or current URL, of the window object.
- `window.document`, which can be shortened to just `document`
  - Returns a reference to the document that the window contains.

Note how we can shorten `window.document` to `document`. For example, the `document` in `document.getElementById('id')` actually refers to `window.document`. All of the methods above can be shortened in the same way.

The browser diagram

We started in the DOM, and we stepped outside it into the BOM. Now, let's take an even higher view of the browser itself. Take a look at this diagram depicting a high-level structure of the browser, from html5rocks.com:

- **User interface**: This is the browser interface that users interact with, including the address bar, back and forward buttons, bookmarks menu, etc. It includes everything except for the requested page content.
- **Browser engine**: Manages the interactions between the UI and the rendering engine.
- **Rendering engine**: Displays, or renders, the requested page content. If the requested content is HTML, it will parse HTML and CSS and render the parsed content.
- **Networking**: Handles network calls, such as HTTP requests.
- **Javascript interpreter**: Parses and executes JavaScript code.
- **UI backend**: Used for drawing basic widgets like combo boxes and windows; uses operating system user interface methods.
- **Data storage**: The persistence of data stored in the browser, such as cookies.

What we learned:

- Review of the DOM, or Document Object Model
- How the BOM differs from the DOM
- The window object and related methods

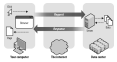
The Request-Response Cycle

Browsing the Web might seem like magic, but it's really just a series of**requests**and **responses**. When we search for information or navigate to a Web page, we are requesting something, and we expect to get a response back.

We can think about the request-response cycle as the communication pattern between a client, or browser, and a server. Whenever we type a URL into the address bar of a browser, we are making a *request*to a server for information back. The most common of these is an `http request`.

The request-response cycle diagram

Let's take a look at this diagram of the request-response cycle fromO'Reily:



On the left is the **client**side, or the browser. On the right is the**server**side, with a database where data is stored. The internet, in the middle, is a series of these client requests and server responses. We'll be going into more depth with servers soon, but for right now we are focusing on the client side.

The browser's role in the request-response cycle

As depicted in the diagram, the browser plays a key role in the request-response cycle. Besides letting the user make the request to the server, the browser also:

1. Parses HTML, CSS, and JS
2. Renders that information to the user by constructing and rendering a DOM tree

When we make a successful request to the server, we are able to view a Web page with content and functionality. Unsuccessful requests prevent the page from loading and displaying information. You've probably seen a 404 page before!

Understanding the request-response cycle is fundamental to developing for the Web. If a server is down, or something is wrong with the request, you'll most likely see an error on the client side. Learning how to debug these errors and set up error handling is a common task for Web developers.

You can go to the **Network tab** of your browser's **Developer Tools** to view these requests and responses. Open a new tab, open up the Developer Tools in your browser, and then navigate to `google.com`. Watch the request go through in your Network tab!

What we learned:

- Reviewed diagram of request-response cycle
- The client side vs. the server side
- The role of the browser
- Where to view Network requests in the browser

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