Lesson | Tuesday

Intermediate JavaScript (/intermediate-javascript) / Object-Oriented JavaScript (/intermediate-javascript/object-oriented-javascript) / Optional: Accessing Stylesheets in the CSSOM

Text

Currently, we know two ways to adjust our website styling from our JavaScript:

- Adjusting the `style` attribute to set inline CSS styles.
- Or adjusting styles by adding and removing `class` attributes that have styles declared for them in our stylesheet.

What we don't know yet is how to access our stylesheet directly, and this is exactly what we'll learn how to do in this lesson.

In the process we'll learn a bit about the CSSOM. Can you guess what CSSOM stands for? The **CSSOM** is the CSS Object Model, another Web API. It's just like the DOM, except instead of representing HTML as an object model, it handles representing our CSS as an object model. The CSSOM is how our web browser makes our project's CSS stylesheet something that we can access and manipulate with JavaScript — by representing the stylesheet as a

series of nested objects. And just like with the DOM, the CSSOM is made up of many object types that we can learn to use in our code, and its data structure is a tree — a hierarchical collection of nodes.

At the end of this lesson, you'll likely still prefer to update your webpage's styles via setting and removing classes. In fact, I also recommend that approach as the easiest to implement. Why? It's because there are a few gotchas involved with accessing our stylesheets:

- Your project may have multiple stylesheets, so you'll need to verify that you are working with the right one.
- We must serve our project (like with VS Code's Live Server extension) to access the stylesheet in order to avoid an issue with the Cross Origin Resource Sharing (CORS) security standard. We'll explain this briefly in this lesson, but cover CORS more in depth when we get to the Asynchrony and APIs course section.
- The list of CSS rules for our stylesheet is a live updated list, which means its contents change as we add and remove rules, which can lead to bugs if we're not careful.

The goal of this lesson is to introduce ourselves to the CSSOM and understand how to access and update our stylesheets. To that end, we'll first start with a review of the Web APIs we've learned about, and see how the CSSOM fits in. Then, we'll get into accessing the right properties and calling the right methods to add and remove CSS rules from our style sheet. We'll learn about:

- The `document.styleSheets` property that contains all of our CSS style sheets (as many as are in our project).
- The `CSSStyleSheets.insertRule()` and `CSSStyleSheets.deleteRule()` methods.
- The `CSSStyleSheets.cssRules` property which contains a live list (updated in real time) of CSS rules.

You are welcome to code along with this lesson, or just read through it. We'll use our Address Book project to practice accessing and updating its stylesheet via the DevTools console.

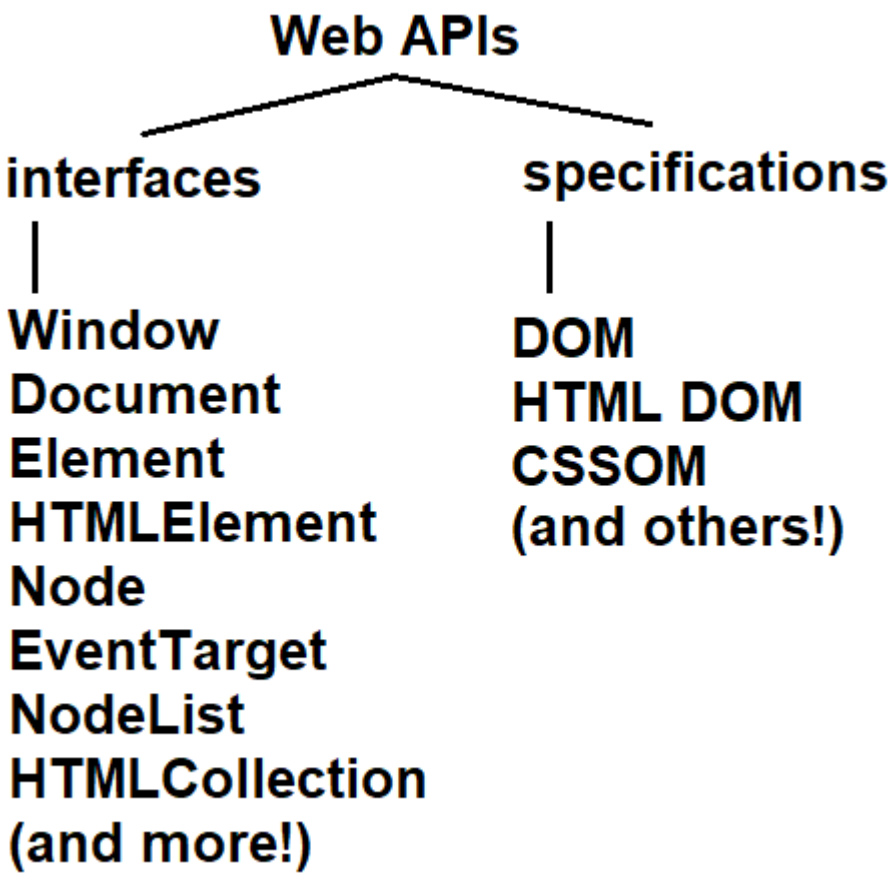
The CSSOM and a Review of Web APIs

The CSSOM is a Web API. Just like the DOM, the CSSOM is a specification that is made up of multiple interfaces (object types). Let's review all of this, starting with what Web APIs are:

- **We must use Web APIs to create interactive webpages.** Web browsers do all the heavy lifting of processing and rendering the webpages we create with HTML, CSS, and JavaScript. Web browsers also manage the events that happen in the browser's webpage like a click of a button or a form submission. So if we want to write JavaScript to respond to events or change the look of our webpage, we must do this by using Web APIs — the web browser structures that are made available to developers so they can create interactive webpages.
- **API is the acronym for application programming interface (API).**
    - A simple example of an interface is a door knob, where the knob abstracts all of the complex inner workings of the latch into a simple tool that we can use to open the door. We don't need to know about how the latch works, we just need to know how to use the knob.
    - Any interface used in computer programming is called an application programming interface, or API. The API lets us access complex and/or private code and use it in our own applications.
    - Web APIs are APIs that specifically make web browser tools and structures available for developers to use in their websites. Here, too, developers don't need to worry about the complex inner workings of web browsers, they just need to learn how to call on the right objects,

methods and properties to respond to an event, or change the webpage.

- **Web APIs have dedicated documentation in MDN.** Visit this link to see the documentation on Web APIs. (https://developer.mozilla.org/en-US/docs/Web/API). These tools are standard across all modern web browsers, however not every tool is implemented in every browser, and some tools are new and experimental.
- **Web APIs are separated into two categories: specifications and interfaces.** A specification describes what functionality or attributes a web tool must include. An example is the DOM. Specifications are typically made up of multiple interfaces. An interface is just an object type that handles one specific piece of functionality. The interfaces we've worked with so far belong to the DOM/HTML DOM: `Window`, `Document`, HTML element objects, `HTMLElement`, `Element`, `EventTarget`, and others.
- **Web browsers create an object model of a website's CSS when it renders the webpage in the browser.** This object model is called the **CSSOM**, and it is made up of objects with properties and methods that we can use in our scripts to access and manipulate our webpage's CSS.
- **The CSSOM specification describes the functionality that the CSS Object Model must include.** When we visit the documentation for the CSSOM on MDN (%5D(https://developer.mozilla.org/en-US/docs/Web/API/CSS_Object_Model)) we can see all of the many interfaces (object types) that make up the functionality of the CSSOM.



It can be overwhelming to wrap your head around all of the object types and specifications out there for Web APIs. Honestly, you really shouldn't know them all by name, or understand how all of the objects relate to each other. What's important to have is a basic conceptual understanding of Web APIs, which we've just outlined above. This will help you be able to better navigate MDN and write search queries, which ultimately will help you become a more resourceful developer.

Beyond conceptually understanding web browser structures and tools (Web APIs), the most important thing here is to be familiar with the object types, methods, and properties that you need to make your code work. So, if the conceptual discussion of Web APIs is a bit too abstract for you, that is completely fine! You don't need to worry about it now. In time, this information will become second nature to you.

Accessing and Updating our CSS Style Sheet(s)

To explore CSSOM tools, we'll use the final version of the Address Book project in the branch `8_adding_delete_functionality_and_polish` (https://github.com/epicodus-lessons/oop-address-book-v2/tree/8_adding_delete_functionality_and_polish). Open your Address Book project in the browser, and then open your DevTools console. Then, input this code:

```
> document.styleSheets;
StyleSheetList {0: CSSStyleSheet, 1: CSSStyleSheet, length:
2}
```

`styleSheets` is a property of the `document` object. The `document.styleSheets` property returns a `StyleSheetList` object that contains one or more `CSSStyleSheet` objects. Each `CSSStyleSheet` object represents one stylesheet and contains the actual CSS rules in that sheet!

The `length` property of the `StyleSheetList` object is always set to a number and it corresponds to how many style sheets are in your project. In our case `length` is set to `2`, which corresponds to the two style sheets in our project: `styles.css` and Bootstrap.

The `StyleSheetList` object is one of those array-like objects we learned about in the "Adding and Removing HTML Elements" lesson in the Arrays and Looping course section. With array-like objects, they have keys that are set to numbers that start at 0 and increment by one. This mimics the index positions of array elements that also start at 0 and increment by one. If we expand the `StyleSheetList` object, we can better see the key-value pairs.

```
StyleSheetList {
  0: CSSStyleSheet {...},
  1: CSSStyleSheet {...},
  length: 2
}
```

Note that even though the keys `0` and `1` appear to be numbers, they are still in fact strings. And when accessing an object key that does not start with an alphabetic character, we must use bracket notation. Let's try using bracket notation to access individual `CSSStyleSheet` objects now:

```
> document.styleSheets[0];
CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rule
s: CSSRuleList, type: 'text/css', href: 'https://cdn.jsdeli
vr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css', …}
> document.styleSheets[1];
CSSStyleSheet {ownerRule: null, type: 'text/css', href: 'fi
le:///C:/Users/staff/Downloads/oop-address-book-v2/css/styl
es.css', ownerNode: link, parentStyleSheet: null, …}
```

There's a lot of information in each object! One helpful property to look at is `href`. The `href` property contains the source of the stylesheet and we can use it to verify which stylesheet is which:

```
> document.styleSheets[0].href;
'https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/boot
strap.min.css'
> document.styleSheets[1].href;
'file:///C:/Users/staff/Downloads/oop-address-book-v2/css/s
tyles.css'
```

This means that the Bootstrap stylesheet is located at the `0` index, and our custom stylesheet `styles.css` is located at the `1` index. Or in other words, the property key for the Bootstrap stylesheet is `0`, and the property key for the `styles.css` stylesheet is `1`.

Note that the value of the `href` property for the `styles.css` stylesheet will look different if you are serving your project with Live Share, or if you are opening your project from a Mac or Linux computer.

Let's pause for a second. Do you have more than just two stylesheets? Maybe the `length` property is set to `3` or `4`? This means that there are additional stylesheets in your project, and this brings us to our first gotcha.

Gotcha #1: Browser Extensions Can Change the Number of CSS Style Sheets in your Webpage

I have a browser extension installed that creates a global dark theme (https://chrome.google.com/webstore/detail/dark-mode/dmghijelimhndkbmpgbldicpogfkceaj?hl=en) that I can toggle on or off for any web page I visit. This extension creates a dark mode on any webpage I visit by actually adding a new style sheet to my webpage. So, when I access `document.styleSheets` after I've served my Mad Libs project, I actually have three `CSSStyleSheet` objects:

```
> document.styleSheets;
< StyleSheetList {0: CSSStyleSheet, 1: CSSStyleSheet, 2: CSSStyleSheet, length: 3}
    0: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, ru
    1: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, ru
    2: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, ru
    length: 3
    [[Prototype]]: StyleSheetList
```

While I might have three stylesheets, you might just have two, or you might have four! The important implication here is that any user accessing our webpage can have any number of stylesheets,

which means that the `styles.css` file that we want to access could be at any location in the list of stylesheets. We need to always keep this in mind during development.

So, the big question here is which stylesheet is `styles.css`?

The easiest way to verify that we're targeting `styles.css` and not another sheet is by looking at the `href` property of each style sheet. Check out the following code:

```
> document.styleSheets[0].href.includes("styles.css");
false
> document.styleSheets[1].href.includes("styles.css");
true
```

By calling `String.prototype.includes()` on the `href` property of each stylesheet object, and passing in the argument `"styles.css"`, we can see if the `href` includes the name of the file we want to target. In this case, we've determined that the stylesheet at index 1 is `styles.css`. This means that the property key of `styles.css` is `1`.

Notably, this may not be true for you. Depending on how many stylesheets you have in your project, and the order in which they are listed, `styles.css` may not be at index 1. **For the purposes of this lesson, we'll always assume that `styles.css` is at index 1 in the examples we work through.** This means that our stylesheet `styles.css` is accessed with `document.styleSheets[1]`.

In order to continually work with the `styles.css` stylesheet, we need to get the name of the key that contains that style sheet so we can access it over and over again, wherever we need to in our code. In our case, the key is `1`.

We also need to be able to check a variable number of stylesheet objects within the `StyleSheetList` collection. To do all of the above, we'll need a loop and branching, and make sure to save the key of

the `styles.css` stylesheet. Our next step is to create a new function that we'll add to our user interface logic. We can try this out in the DevTools first:

```
> function locateStyleSheet() {
  const styleSheetArray = Array.from(document.styleSheets);
  let indexLocation;
  styleSheetArray.forEach(function(styleSheet, index) {
    if (styleSheet.href) {
      if (styleSheet.href.includes("styles.css")) {
        indexLocation = index;
      }
    }
  });
  return indexLocation;
}
> locateStyleSheet();
1
```

Let's break down the new `locateStyleSheet()` function:

- First we turn the `StyleSheetList` object into an array by passing `document.styleSheets` into the `Array.from()` static method, and saving the result in the variable `styleSheetArray`. This will enable us to use array methods on the list of stylesheets.
- Next, we initialize an empty variable called `indexLocation`. This is the variable that will contain the property key of the `styles.css` stylesheet object. At the end of the function, we return `indexLocation` from our function.
- Next, we loop over our `styleSheetArray` array with `Array.prototype.forEach()`. Remember that `Array.prototype.forEach()` takes a callback function as an argument. We're passing in an anonymous function expression as the callback, which is highlighted below:

```
// this is the callback function
// passed into styleSheetArray.forEach(...);
function(styleSheet, index) {
  if (styleSheet.href) {
    if (styleSheet.href.includes("styles.css")) {
      indexLocation = index;
    }
  }
}
```

- The callback function for `Array.prototype.forEach()` has access to two optional parameters: one that gives us access to the current element in the array (the `styleSheet` parameter), and the other that gives us access to the current element's index location (the `index` parameter). Often these parameters are call `element` and `index`, but we can name them whatever we want. We're using both parameters so that we can access each stylesheet (the elements in our array) and their index location (which corresponds to their property key in the `StyleSheetList` object).
- Within the callback function, we have an `if` statement that first checks to see if there's any value for the current stylesheet's `href` property. We're harnessing JavaScript's concept of truthy values, and we're using it to weed out any stylesheets that have `null` value for the `href` property. Why? If we try to call a string method on `null`, JavaScript will throw an error and stop the execution of our code. So, if our styleSheet has an `href` property with any value, we then move onto the second, nested `if` statement.
- In the second, nested `if` statement, we check whether the stylesheet's `href` property includes `"styles.css"`, and if so to update the value of our `indexLocation` variable with the index. If the stylesheet's `href` property does not have `"styles.css"` in it, we do nothing and the loop continues.

Let's see how we can actually put the `locateStyleSheet()` function to use. In the DevTools, we'll call the `locateStyleSheet()` function and saved the returned result in a variable, and then we'll use that variable to access `document.styleSheets`:

```
> const indexOfCustomStyles = locateStyleSheet();
> indexOfCustomStyles;
1
> document.styleSheets[indexOfCustomStyles];
CSSStyleSheet {ownerRule: null, type: 'text/css', href: 'fi
le:///C:/Users/staff/Downloads/oop-address-book-v2/css/styl
es.css', ownerNode: link, parentStyleSheet: null, …}
```

Notably, `document.styleSheets[indexOfCustomStyles]` returns the same result as `document.styleSheets[1]`, since `indexOfCustomStyles` is set to `1`.

Now with the functionality of the `locateStyleSheet()` function, we can quickly find the key (or index location) of our custom style sheet `styles.css`, no matter how many other stylesheets there are. The `locateStyleSheet()` function specifically enables us to find the index of `styles.css` **dynamically**. When we find values and results dynamically, we avoid using hardcoded values, and instead use algorithms to generate the values we need.

Just to reiterate, here's a hardcoded index:

```
// hardcoded index
> document.styleSheets[1];
```

And here's a dynamically generated index:

```
// dynamic index
> const indexOfCustomStyles = locateStyleSheet();
> indexOfCustomStyles;
> document.styleSheets[indexOfCustomStyles];
```

Next up, let's explore how to access the rules in our `styles.css` stylesheet.

Accessing the Rules in our Stylesheet

Now that we have access to our stylesheet, let's access the rules inside of it. To do this, we'll use properties and methods from the `CSSStyleSheet` object, which will let us look at all of the rules listed in our style sheets, as well as delete and insert rules:

- The `CSSStyleSheet.cssRules` property lets us access all of the rules listed in the style sheet.
- `CSSStyleSheet.deleteRule()` method lets us remove a CSS rule from the style sheet.
- `CSSStyleSheet.insertRule()` method lets us add a CSS rule from the style sheet.

We'll start by looking at all of the rules in the `CSSStyleSheet.cssRules` property. Remember that in this lesson's examples, we're assuming that the `styles.css` stylesheet is at index 1 (or, its property key is `1`) and can be accessed with `document.styleSheets[1]`.

We can access the `cssRules` property in a couple of ways. First by accessing the `styles.css` stylesheet, and then the rules:

```
> const myCustomStyles = document.styleSheets[1];
> myCustomStyles.cssRules;
```

Or by accessing the `cssRules` by chaining property accessors:

```
> document.styleSheets[1].cssRules;
```

And what do we get?

```
> document.styleSheets[1].cssRules;
Uncaught DOMException: Failed to read the 'cssRules' proper
ty from 'CSSStyleSheet': Cannot access rules at <anonymous
>:1:25
```

Uh-oh! We get a `DOMException` error. Well, you'll only get this error if your project is not being served by any web server, like Live Server. This brings us to our next gotcha.

Gotcha #2: You Must Serve your Project to Access the `cssRules` Property of your Stylesheet

This gotcha is caused by two things. First of all, the `cssRules` property is live-updated. This means that it's not just static data, but data that's continuously being updated. In other words, any time we add or remove a rule from our style sheets, the `cssRules` property automatically reflects this change.

The second reason behind the `DOMException` error has to do with the Cross Origin Resource Sharing (CORS) security standard that all modern browsers implement. We will spend time learning about CORS when we query APIs with asynchronous JavaScript in the 6th course section of the program, so we won't get into the weeds of understanding CORS right now. We'll just focus on incorporating the solution in our projects so we can start accessing and updating our CSS rules.

The solution for accessing the `CSSStyleSheet.cssRules` property is to simply serve our project with Live Share. Even when you are working alone, you'll need to serve your project with Live

Share to access the `CSSStyleSheet.cssRules` property. If we do not serve our project, the CORS security standard will prevent the `cssRules` property from getting an updated rule list.

When we serve our project with LiveShare or otherwise, the URL will come from `localhost`. For my Address Book project, the URL looks like `http://127.0.0.1:5501/index.html`. In my case, `127.0.0.1` is the same as `localhost`, so if I am using Live Server to serve my project, I can also navigate to `http://localhost:5501/index.html` to access it.

The opposite of serving a project is just opening a local file directly in the browser. For example, on my Windows computer, if I open my Address Book html file in the browser, the URL will state this `file:///C:/Users/staff/Desktop/oop-address-book-v2/index.html`. A URL that starts with `file:///` or `C:/Users/...` indicates that we've simply opened a local file in the browser from a Windows computer, and we're not serving the project with a web server.

If you are coding along with this lesson, make sure that your project is opened in the browser with Live Server before continuing.

Accessing our Rules Again

With our project served, we can now access `cssRules` to view the rules in `styles.css`. We'll also add a few new ones for practice. Again, in this lesson, we're assuming that `styles.css` is located at index 1 of `document.styleSheets`.

```
> document.styleSheets[1].cssRules;
CSSRuleList {0: CSSStyleRule, length: 1}
  0: CSSStyleRule {selectorText: '.hidden', style: CSSStyle
Declaration, styleMap: StylePropertyMap, type: 1, cssText:
'.hidden { display: none; }', …}
  length: 1
  [[Prototype]]: CSSRuleList
```

The `cssRules` property returns a `CSSRuleList` object which contains a collection of `CSSStyleRule` objects, each of which represents an individual rule.

Or in other words, every CSS rule in any stylesheet object is turned into a `CSSStyleRule` object, and added to the list of CSS rules that's saved in the `cssRules` property, which is a `CSSRuleList` object.

Our `styles.css` stylesheet only has one rule:

```
.hidden {
  display: none;
}
```

Which means that our `style.css` stylesheet object only has one `CSSStyleRule` object, which we can access with bracket notation:

```
> document.styleSheets[1].cssRules[0];
CSSStyleRule {selectorText: '.hidden', style: CSSStyleDecla
ration, styleMap: StylePropertyMap, type: 1, cssText: '.hid
den { display: none; }', …}
```

To access the exact text of our `.hidden` rule, we'll need to do access its `cssText` property:

```
> document.styleSheets[1].cssRules[0].cssText;
'.hidden { display: none; }'
```

Adding Two New CSS Rules with `CSSStyleSheet.insertRule()`

Let's add a new rule to our `styles.css` stylesheet. To do this, we'll call the `CSSStyleSheet.insertRule()` method, which is called on the stylesheet object, and not on its list of CSS rules found in the `cssRules` property.

```
> const myCustomStyles = document.styleSheets[1]
> myCustomStyles.insertRule('body { background-color: red
}');
```

The `CSSStyleSheet.insertRule()` method takes a string as an argument that contains the CSS rule. And we've done just that: take a CSS rule and add quotes around to make it a string.

We'll know that we've successfully added a new rule if the background color of our Address Book project turns red! We can also check that our `cssRules` property now has two rules listed:

```
> document.styleSheets[1].cssRules;
CSSRuleList {0: CSSStyleRule, 1: CSSStyleRule, length: 2}
  0: CSSStyleRule {selectorText: 'body', style: CSSStyleDec
laration, styleMap: StylePropertyMap, type: 1, cssText: 'bo
dy { background-color: red; }', …}
  1: CSSStyleRule {selectorText: '.hidden', style: CSSStyle
Declaration, styleMap: StylePropertyMap, type: 1, cssText:
'.hidden { display: none; }', …}
  length: 2
  [[Prototype]]: CSSRuleList
```

Notice that our new CSS rule has been added to the top of the list of CSS rules, at index 0. This is the default behavior of the `CSSStyleSheet.insertRule()` method. However, we can include an

optional second argument to specify which index location we want to add the rule. Let's try that out next.

We'll add our next rule to the very end of the list of CSS rules:

```
> const endOfListIndex = document.styleSheets[1].cssRules.l
ength;
> document.styleSheets[1].insertRule('div.container { colo
r: green; font-weight: 600 }', endOfListIndex);
```

First, we accessed the length of the `cssRules` property (which corresponds to how many rules it has), and then we've passed that in as the second argument in the `CSSStyleSheet.insertRule()` method call. This adds the new rule for our div with the `.container` class at the end of our list of rules.

```
> document.styleSheets[1].cssRules;
CSSRuleList {0: CSSStyleRule, 1: CSSStyleRule, 2: CSSStyleR
ule, 3: CSSStyleRule, length: 3}
  0: CSSStyleRule {selectorText: 'body', style: CSSStyleDec
laration, styleMap: StylePropertyMap, type: 1, cssText: 'bo
dy { background-color: red; }', …}
  1: CSSStyleRule {selectorText: '.hidden', style: CSSStyle
Declaration, styleMap: StylePropertyMap, type: 1, cssText:
'.hidden { display: none; }', …}
  2: CSSStyleRule {selectorText: 'div.container', style: CS
SStyleDeclaration, styleMap: StylePropertyMap, type: 1, css
Text: 'div.container { color: green; font-weight: 600; }',
…}
  length: 3
  [[Prototype]]: CSSRuleList
```

Next up, let's explore how to remove rules.

Removing Rules with `CSSStyleSheet.deleteRule()`

We use the `CSSStyleSheet.deleteRule()` method. This method take an index as an argument. Let's delete the red background color. First, let's verify the index location of this rule:

```
> document.styleSheets[1].cssRules;
CSSRuleList {0: CSSStyleRule, 1: CSSStyleRule, 2: CSSStyleR
ule, 3: CSSStyleRule, length: 3}
  0: CSSStyleRule {selectorText: 'body', style: CSSStyleDec
laration, styleMap: StylePropertyMap, type: 1, cssText: 'bo
dy { background-color: red; }', …}
  1: CSSStyleRule {selectorText: '.hidden', style: CSSStyle
Declaration, styleMap: StylePropertyMap, type: 1, cssText:
'.hidden { display: none; }', …}
  2: CSSStyleRule {selectorText: 'div.container', style: CS
SStyleDeclaration, styleMap: StylePropertyMap, type: 1, css
Text: 'div.container { color: green; font-weight: 600; }',
…}
  length: 3
  [[Prototype]]: CSSRuleList
```

We should be able to tell that the background color is set in the rule at index 0. With that information we can remove the rule:

```
> document.styleSheets.deleteRule(0);
undefined
```

The `CSSStyleSheets.deleteRule()` method doesn't return anything, so the console prints `undefined`.

We should be able to verify that our rule deletion worked by the change in background color of our Address Book project. Now it should be white again.

However, let's double-check our `cssRules` property to verify as well. This will reveal the next gotcha!

Gotcha #3: The Changing Index Locations of CSS Rules Can Lead to Unexpected Bugs

```
> document.styleSheets[1].cssRules;
CSSRuleList {0: CSSStyleRule, 1: CSSStyleRule, 2: CSSStyleR
ule, 3: CSSStyleRule, length: 3}
  0: CSSStyleRule {selectorText: '.hidden', style: CSSStyle
Declaration, styleMap: StylePropertyMap, type: 1, cssText:
'.hidden { display: none; }', …}
  1: CSSStyleRule {selectorText: 'div.container', style: CS
SStyleDeclaration, styleMap: StylePropertyMap, type: 1, css
Text: 'div.container { color: green; font-weight: 600; }',
…}
  length: 2
  [[Prototype]]: CSSRuleList
```

Notice how the keys for our CSS rules have shifted! The `.hidden` rule is now at index 0, and the `div.container` rule is now at index 1. This is another gotcha when adding and removing CSS rules: the index locations of individual rules change when we add and remove rules. This happens because `cssRules` is a live updated property that always shows an updated list, which includes updates index locations for CSS rules as new ones get added, and others are removed.

The implication for us developers is that this makes targeting the right rule a bit more cumbersome of a process, because we can't hardcode index values. Instead, we need to generate them dynamically. For example, we might need to write a similar function as we did with the `locateStyleSheet()` function to make sure we're targeting the right rule at the right index location.

Or, we could be very diligent about only adding and removing new rules to the beginning of the list at index 0, so that we could still hardcode. However, this would only be viable if we're adding and removing just one rule, and it's not code that is very scalable. What happens if we need to add and remove multiple rules? Well, then we're back to needing a solution similar to the `locateStyleSheet()` function.

In the end, the lesson here is that these changing indices are something that we have to consider as a developer. In fact, all of the gotchas we learned about are factors that we need to consider when we're working with the CSSOM to directly access and update our stylesheets.

A Review of the CSSOM Object Types and Properties Names

With a statement like this:

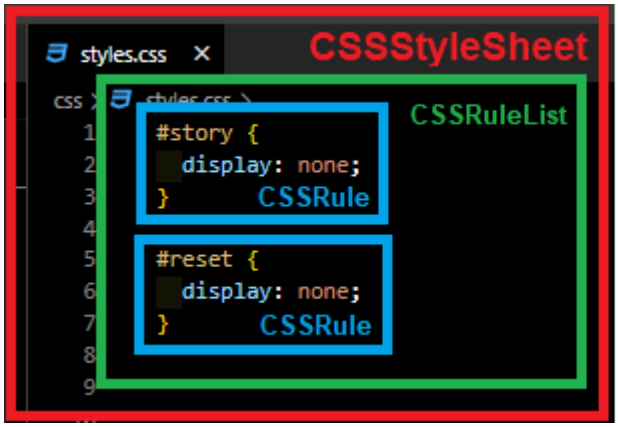
```
> document.styleSheets[1].cssRules[0].cssText;
```

We've possibly gone farther than we ever have in a series of nested objects. If you are anything like me, this is definitely confusing.

Also, with the CSSOM being so new to us, we're learning the properties and methods we need to call, but we're also learning about new object types within the CSSOM. Differentiating between the property names and the object types is also confusing.

So, let's do a review of the new objects types and properties that we've worked with in this lesson.

Check out the following image that shows CSSOM object types as they relate to an actual CSS stylesheet. Take note that the CSS stylesheet in the image below is fictional, and does not belong to the Address Book project.



In the image to the right, we can see how all of these object types nest inside of each other. In the image on the left, we see a series of nested boxes that visualize what each object type corresponds to in the CSS stylesheet.

The `StyleSheetList` object contains a list of all stylesheets in a project.

- Each stylesheet in the `StyleSheetList` is represented by a `CSSStyleSheet` object.
- We access the `StyleSheetList` via `document.styleSheets`.

The `CSSStyleSheet` object represents a single CSS stylesheet.

- We access a `StyleSheetList` via `document.styleSheets[index]`, where `index` is the index of the stylesheet we want to target.
- We can access various helpful properties and methods from this object:
  - `CSSStyleSheet.cssRules` contains a list of all CSS rules in the stylesheet.
  - `CSSStyleSheet.href` contains the path to the location of the stylesheet.
  - `CSSStyleSheet.insertRule()` is a method to add a rule to the list of CSS rules.
  - `CSSStyleSheet.deleteRule()` is a method to remove a rule from the list of CSS rules.

The `CSSRuleList` object contains a list of CSS rules in a stylesheet.

- Each CSS rule in the `CSSRuleList` is represented by a `CSSRule` object.
- We access the `CSSRuleList` via `document.styleSheets[index].cssRules`, where `index` is the index of the stylesheet we want to target, and `cssRules` is a property of the `CSSStyleSheet` object.

The `CSSRule` object represents a single CSS rule.

- We access the `CSSRule` via `document.styleSheets[index1].cssRules[index2]`, where `index1` is the index of the stylesheet we want to target, and `index2` is the index of the single CSS rule that we want to target.
- We can access various properties in the `CSSRule` object, like:
  - `CSSRule.cssText` which contains the CSS rule as a string.

Summary

In this lesson we learned how to access the CSSOM, including how to view, add, and remove CSS rules. In the process, we reviewed browser Web APIs and learned how the CSSOM fits in. We also learned a few important gotchas when writing code that accesses a specific stylesheet and updates specific CSS rules:

- Users who visit your site may have additional stylesheets that are added by browser extensions. So, you'll need to be sure to locate your custom stylesheet dynamically, and not hard code an index value that could change.
- In order to access the live-updated list of CSS rules in your stylesheet, you must serve your project (like with VS Code's Live Server extension) in order to avoid an issue with the Cross Origin Resource Sharing (CORS) security standard. (We'll learn

more about this standard in the Asynchrony and APIs course section).
- The list of CSS rules for our stylesheet is a live updated list, which means its contents change as we add and remove rules. The big implication for us is that the indexes of the contents change, and we use these indexes to remove rules, and sometimes when adding rules. Here, too, we can't always rely on hardcoded indexes in our code and expect it to function without error.

As stated at the very beginning of this lesson, because of these gotchas it's much easier and just as effective to add and remove CSS styling by adding and removing classes with `Element.setAttribute()` and `Element.removeAttribute()`. It will work for the majority of our use cases. But the truth is that we've only covered one small aspect of the CSSOM by looking at `document.styleSheets`, and there are more tools to learn about for manipulating the CSSOM.

If you are an aspiring front-end developer, I recommend that you continue research on the CSSOM and what more you can do with it. Here are a few resources to work through:

- "An Introduction and Guide to the CSS Object Model (CSSOM)" by css-tricks.com (https://css-tricks.com/an-introduction-and-guide-to-the-css-object-model-cssom/)
- "Populating the page: how browsers work" by MDN (https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work)
- The CSSOM Specification on MDN (https://developer.mozilla.org/en-US/docs/Web/API/CSS_Object_Model)

If you want a challenge, try updating the show/hide functionality for contact details in the Address Book project to directly add and remove the `.hidden` rule from the project's `styles.css` stylesheet.

Previous (/intermediate-javascript/object-oriented-javascript/introduction-to-the-node-object)
Next (/intermediate-javascript/object-oriented-javascript/game-of-choice-two-day-project)

Lesson 25 of 33
Last updated March 23, 2023

disable dark mode

(http://www.epicodus.com)