Exploration — Modifying the DOM Tree

Introduction

In this exploration we continue our study of DOM tree. Specifically, we now turn our attention to how the DOM API can be used to modify both the contents as well as the structure of a DOM tree. We will study the low-level JavaScript API to do this. We again note that the modification of a DOM tree is now typically done with high-level APIs that we will study later in this course. But a conceptual understanding of the low-level API will help our understanding of what exactly is being done under the hood by the high-level APIs. This will help you understand different high-level APIs, including those we do not discuss in this course, as well as those that have not been invented yet!

Commonly Used Properties of Nodes

textContent

Probably the most basic property of a node is its `textContent`. This contains all the text content of a node, and its children, in a string representation. Setting a node's `textContent` to an empty string will clear out all of its text and that of its child nodes.

innerHTML

Nodes of type element have a property named `innerHTML` which get the HTML markup within an element.

Example: textContent vs. innerHTML

Consider the following HTML element.

```
<p id="book">I am reading a <b>JavaScript</b> book. You can find it <a href="https://www.amazo
n.com/Modern-JavaScript-Impatient-Cay-Horstmann/dp/0136502148">here</a>.
</p>
```

We can log the `textContent` and `innerHTML` of this `p` element as follows:

```
let text = document.getElementById('book').textContent;
console.log(text);

let html = document.getElementById('book').innerHTML;
console.log(html);
```

Here is what will be printed to the browser log:

```
I am reading a JavaScript book. You can find it here.
I am reading a <b>JavaScript</b> book. You can find it <a href="https://www.amazon.com/Modern-JavaScript-Impatient-Cay-Horstmann/dp/0136502148">here</a>.
```

As we can see `textContent` of this `p` element contains its text content and that of its child element `b` concatenated as a string, but it does not contain the HTML tags or the attribute. On the other hand, the `innerHTML` property contains all the HTML content, including the tags of child elements and attributes inside the element `book`.

Note: Using the property `innerHTML` to insert text into a webpage is a security risk. It is recommended that the insertion of plain text should be done using `textContent` and should not use `innerHTML`. See the **discussion at MDN (https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML#security_considerations)** for an explanation.

style

Another important property of a node is its `style` property which lets us change the style of the node. In the JavaScript DOM API, the style property is represented as a JavaScript object. The property names and values of this JavaScript object correspond to the CSS property names and values of the styles associated with this node in the DOM. For example, `myNode.style.color = "red";` will style `myNode` to have `red` text. Some CSS property names like `background-color` include a dash. In JavaScript, such property names are converted to camel case names with the dash removed and the letter after the dash being capitalized. Thus to change the background color of a node to violet, we use `myNode.style.backgroundColor = "violet";`.

className

For element nodes and important property is the `className` property which gives us access to that elements' class name. This property can hold multiple class names that are separated by spaces. If we wanted to append an additional class, say `newClass` to an element `myElement`, we can do that by using `myElement.className += ' newClass'`. Note the space before `newClass`. Without it, the `className` property will have just one class named `oldClassnewClass`.

Modifying the Structure of DOM Trees

Adding Nodes

Adding nodes in a DOM tree is a two step process:

1. Create the node by calling the appropriate method on the `document` object.
2. The newly created node does not yet appear in the DOM tree. As the second step, we insert the new node at the desired position in the DOM tree.

Creating Nodes

The DOM API does not provide one general method that can create nodes of different types. Instead, the API provides different methods on the `document` object that can create nodes of specific types. We look at two of these methods, the ones that create text nodes and element nodes.

Creating Text Nodes

The method **document.createTextNode() (https://developer.mozilla.org/en-US/docs/Web/API/Document/createTextNode)** can be used to create new text nodes. However, in most cases we can simply add text content using the `textContent` property we discussed earlier, rather than creating a text node.

Creating Element Nodes

To create elements we use the method **document.createElement() (https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement)**. It takes one argument which is the tag of the element we want to create and returns an element of that type. However, the newly created element does not yet exist in the DOM tree.

For example, `let newPara = document.createElement('p');` will create a new `p` element node which will be held in the variable `newPara`.

Inserting Nodes

There are many different methods available to insert nodes in a DOM tree, but one of the most common ways to insert elements is calling the method **appendChild(newChild) (https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild)** on a node. This method will add the node `newChild` as the last child of the node the method was called on.

For example, the statement `document.getElementById('bigDiv').appendChild(newPara);` will append the element `newPara` as the last child of the element with an id of `bigDiv`.

Removing Nodes

To remove an element, we first need to get a reference to the element and its parent. Then on the parent node we call the method **removeChild (https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild)** and pass it the node we want to remove as an argument.

Example Modifying the Structure of DOM Trees

In the following example, we use the DOM API in our JavaScript code to create an ordered list of three elements and add this list to an existing `div` element in the document. We then use the `style` property of the list elements to give a different style to each of the three list elements.

index.html ×

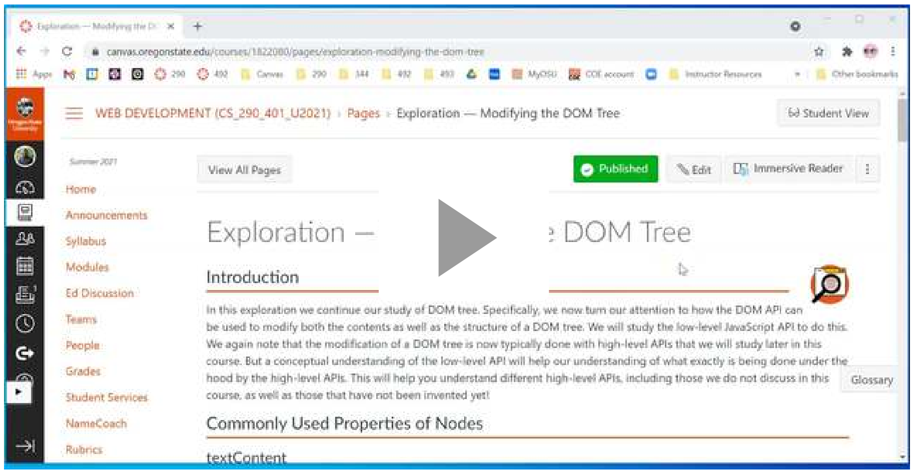
```html
<!doctype html>
<!doctype html>
<html>

<head>
    <title>Creating Nodes</title>
    <script src="script.js"></script>
    <link href="style.css"
rel="stylesheet" type="text/css" />
</head>
```

```html
></div>
```

Code Editor ×

This is where you write code. Code describes how programs work. The editor helps you write code, by coloring certain types, and giving you suggestions when you type. Click on one of the examples to see how code looks.

› Next


Exploration — Modifying the D ×
canvas.oregonstate.edu/courses/1822080/pages/exploration-modifying-the-dom-tree
WEB DEVELOPMENT (CS_290_401_U2021) › Pages › Exploration — Modifying the DOM Tree
Student View
Summer 2021
Home
Announcements
Syllabus
Modules
Ed Discussion
Teams
People
Grades
Student Services
NameCoach
Rubrics
View All Pages
Published
Edit
Immersive Reader
Glossary

Exploration — … DOM Tree

Introduction

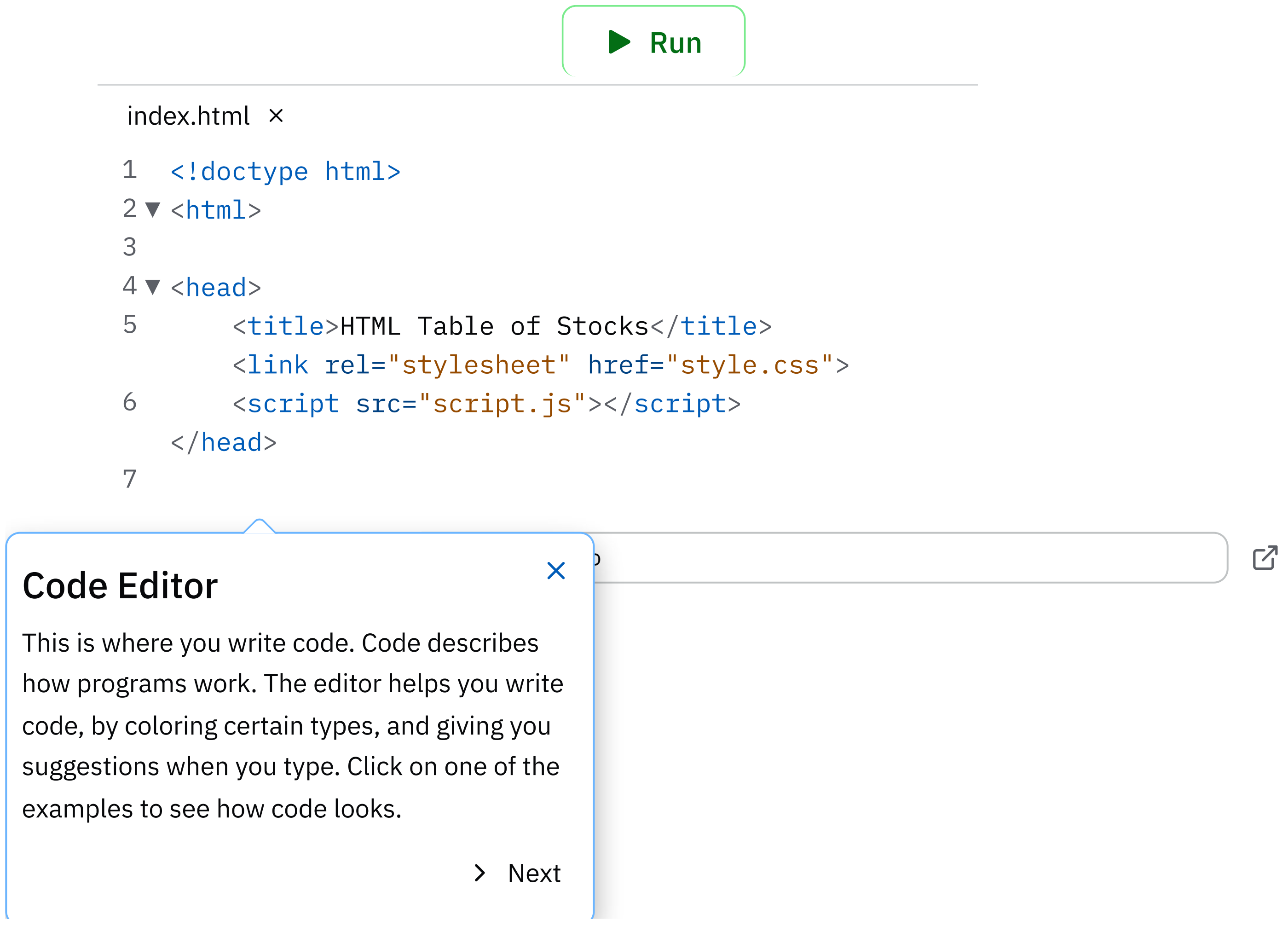
In this exploration we continue our study of DOM tree. Specifically, we now turn our attention to how the DOM API can be used to modify both the contents as well as the structure of a DOM tree. We will study the low-level JavaScript API to do this. We again note that the modification of a DOM tree is now typically done with high-level APIs that we will study later in this course. But a conceptual understanding of the low-level API will help our understanding of what exactly is being done under the hood by the high-level APIs. This will help you understand different high-level APIs, including those we do not discuss in this course, as well as those that have not been invented yet!

Commonly Used Properties of Nodes

textContent

Exercise

Add JavaScript code in the file `scripts.js` to populate the HTML table with one row corresponding to each stock in the object `stocks`. Places where you need to add code are marked with the comment `// TODO: ADD CODE`



```
<!doctype html>
<html>

<head>
    <title>HTML Table of Stocks</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
</head>

```

Code Editor

This is where you write code. Code describes how programs work. The editor helps you write code, by coloring certain types, and giving you suggestions when you type. Click on one of the examples to see how code looks.

Summary

At this point we can start using JavaScript to manipulate and modify the DOM tree. We know how to access and modify properties of existing nodes in a DOM tree. We also know how to change the structure of a DOM tree by adding and removing nodes from the DOM tree. However, to create fully interactive pages, we need to understand and use DOM events which we will study next.

Additional References

Here are some references to learn more about the topics we discussed in this exploration.

- Methods to create nodes of different type are defined on the [Document node in the DOM (https://developer.mozilla.org/en-US/docs/Web/API/Document)](https://developer.mozilla.org/en-US/docs/Web/API/Document) and typically start with the string `create`.
- [Traversing an HTML table with JavaScript and DOM Interface (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Traversing_an_HTML_table_with_JavaScript_and_DO](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Traversing_an_HTML_table_with_JavaScript_and_DO)

M_Interfaces) at MDN has very good examples showing the use of the DOM API for traversing and modifying the structure of a DOM tree, as well as examples of modifying the styling of nodes in the tree.