SUNSTONE

JavaScript

LECTURE 15 NOTES

The Document Object Model

- **The Document Object Model (DOM)**

  - The document object represents the whole HTML document.
  - When an HTML document is loaded in the browser, it becomes a document object. It is the root element that represents the HTML document. It has properties and methods. By the help of document objects, we can add dynamic content to our web page.
  - DOM is the object of the window. So, `window.document` is the same as `document`.

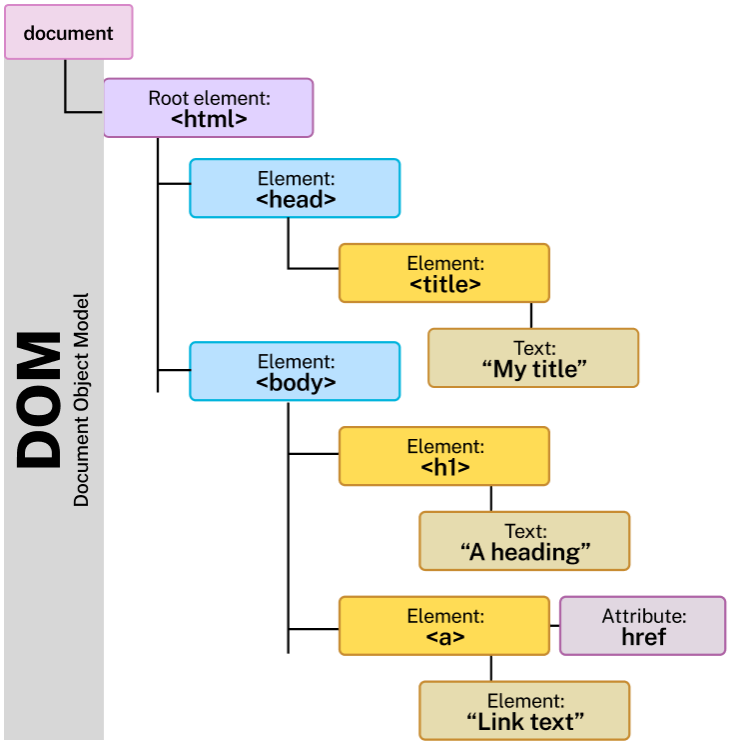
  According to W3C (World Wide Web Consortium) -

  - "The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document".
  - With the object model, JavaScript gets all the power it needs to create dynamic HTML:
    - JavaScript can change all the HTML elements in the page
    - JavaScript can change all the HTML attributes in the page
    - JavaScript can change all the CSS styles in the page
    - JavaScript can remove existing HTML elements and attributes
    - JavaScript can add new HTML elements and attributes
    - JavaScript can react to all existing HTML events in the page
    - JavaScript can create new HTML events in the page

  Note:

  - The DOM is a standard for how to get, change, add, or delete HTML elements.
  - The DOM is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree-like model, where each node in the tree represents an element, attribute, or text within the document.

- **The Structure and Hierarchy of the DOM**
  - The DOM is a programming interface for web documents. It represents the structure and content of an HTML, XHTML, or XML document as a tree-like structure, where each node in the tree represents an element, attribute, or piece of text in the document.
  - The DOM structure starts with the root node, which represents the entire document. From the root node, the tree branches out into child nodes, representing the various elements and their hierarchical relationships within the document.
  - Here is a simplified representation of the DOM structure:
    - **Document (root node)**
    - **Element nodes (represent HTML elements)**
      - Attribute nodes (represent attributes of elements)
      - Text nodes (represent text content within elements)
      - Event handler nodes (represent event handlers attached to elements)
    - **Comment nodes (represent comments within the document)**
    - **Processing Instruction nodes (represent processing instructions)**
    - **Document Type Declaration node (represents the document type declaration)**



- Each element in the DOM tree is represented by an **element node**, which may have **child nodes** and can also have attributes associated with it. The attributes are represented by **attribute nodes**, which contain the attribute name and its corresponding value.
- Text content within elements is represented by **text nodes**, which store the actual textual content. **Event handler nodes** store info about event handlers attached to elements, such as JavaScript code to be executed when a specific event occurs.
- **Comment nodes** represent comments within the document and can be used to provide additional information or annotations. **Processing Instruction nodes** represent processing instructions, which are typically used in XML documents to provide instructions for the application processing the document.
- The **Document Type Declaration node** represents the document type declaration, which specifies the type of document (e.g., HTML, XHTML) and its associated rules.

➢ **Key Nodes and their Hierarchical Relationships:**

- **Document Nodes**
  - Represents the entire HTML or XML document
  - Acts as the entry point to access the document's contents
  - Contains all other nodes in the document
- **Element Nodes**
  - Represent HTML or XML elements (tags) such as , <p>, <h1>, etc.
  - Forms the main building blocks of the document structure
  - Can contain other elements, text, or other types of nodes
  - Have attributes (e.g., id, class) that provide additional information about the element
- **Text Nodes**
  - Represent the actual text content within an element
  - Can be accessed and modified using DOM methods and properties

- **Attribute Nodes**
  - Represent the attributes of an element
  - Hold additional information about the element
  - Accessed through the element nodes
- **Comment Nodes**
  - Represent HTML or XML comments
  - Contain text that is not rendered but provides information or notes
- **Document Type Nodes**
  - Represents the document type declaration (<!DOCTYPE>)

- The hierarchy of these nodes forms a tree-like structure, with the document node at the root and the other nodes branching out as children, grandchildren, and so on. Elements can have multiple children, which can be other elements, text nodes, or other types of nodes.
- The DOM hierarchy allows you to traverse the structure, access individual nodes, modify their content or attributes, create new nodes, and remove existing nodes. This flexibility enables dynamic manipulation and interaction with web pages through JavaScript or other programming languages.

- **Accessing DOM Elements**

1. **By ID:** Use the getElementById() method, which retrieves an element by its unique ID attribute.

```
const element = document.getElementById('elementId');
```

2. **By class name:** Use the getElementsByClassName() method, which returns a collection of elements with the specified class name.

```
const elements =
```

```
document.getElementsByClassName('className');
```

3. **By tag name:** Use the getElementsByTagName() method, which returns a collection of elements with the specified tag name.

```
const elements = document.getElementsByTagName('tagName');
```

4. **By CSS selector:** Use the querySelector() or querySelectorAll() method to select elements using CSS selector syntax.

```
const element = document.querySelector('selector');
```

```
const elements = document.querySelectorAll('selector');
```

- **Manipulating DOM Elements**
  - Once you have accessed a DOM element, you can manipulate its properties, attributes, or content.
    - **Changing text content:** Use the textContent property to change the text content of an element.

```
element.textContent = 'New text content';
```

    - **Changing HTML content:** Use the innerHTML property to change the HTML content of an element.

```
element.innerHTML = '<strong>New HTML content</strong>';
```

    - **Changing attributes:** Use the setAttribute() method to set or modify an attribute of an element.

```
element.setAttribute('attributeName', 'attributeValue');
```

    - **Adding or removing CSS classes:** Use the classList property to add or remove CSS classes from an element.

```
element.classList.add('className');
```

```
element.classList.remove('className');
```

- **Modifying style properties:** Use the style property to modify CSS styles of an element.

```
element.style.property = 'value';
```

- **Adding event listeners:** Use the addEventListener() method to attach event handlers to elements.

```
element.addEventListener('eventName', eventHandler);
```

- **Events and Event Handling**
  - Events in JavaScript represent actions or occurrences that happen in the browser, such as a button click, mouse movement, or keyboard input.
  - Event handling involves writing code to respond to these events. Here are some examples of events and event handling in JavaScript:

1. **Click Event:** This event is triggered when an element is clicked.

```
const button = document.getElementById('myButton');
button.addEventListener('click', function(event) {
// Event handling code
console.log('Button clicked!');
});
```

2. **Mouseover and Mouseout Events:** These events are triggered when the mouse pointer enters or leaves an element.

```
const element = document.getElementById('myElement');
element.addEventListener('mouseover', function(event) {
// Event handling code
console.log('Mouse over the element!');
});
element.addEventListener('mouseout', function(event) {
// Event handling code
console.log('Mouse out of the element!');
```

```
});
```

3. **Keypress Event:** This event is triggered when a key is pressed while the focus is on an element.

```
const input = document.getElementById('myInput');
input.addEventListener('keypress', function(event) {
// Event handling code
console.log('Key pressed!');
});
```

4. **Form Submit Event:** This event is triggered when a form is submitted.

```
const form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
// Prevent the default form submission
event.preventDefault();
// Event handling code
console.log('Form submitted!');
});
```

5. **Window Load Event:** This event is triggered when the entire page has finished loading, including all external resources.

```
window.addEventListener('load', function(event) {
// Event handling code
console.log('Page loaded!');
});
```

6. **Event Delegation:** Event delegation allows you to handle events on dynamically added or multiple elements by delegating the event handling to a parent element.

```
const parentElement = document.getElementById('parent');
parentElement.addEventListener('click', function(event) {
  // Check if the clicked element matches a specific selector
  if (event.target.matches('.child')) {
    // Event handling code for the child element
    console.log('Child element clicked!');
  }
});
```

- **Performance and Optimization Considerations**

  - **Minimize DOM Access:** DOM operations can be relatively expensive, so it's crucial to minimize unnecessary DOM access. Each time you access or manipulate the DOM, there is a potential performance cost. Instead of accessing the DOM repeatedly, consider caching the references to DOM elements in variables to reuse them.

  - **Batch DOM Modifications:** When making multiple DOM modifications, such as updating multiple styles or appending multiple elements, it's more efficient to batch those modifications together. This reduces the number of reflows and repaints triggered by the browser. You can accomplish this by using techniques like Document Fragments or manipulating elements outside the DOM flow before reinserting them.

  - **Limit Reflows and Repaints:** Changes to the DOM can trigger reflows and repaints, which can be expensive operations. Minimize the number of changes that require the browser to recalculate layout and repaint the screen. For example, avoid making frequent changes to element styles within a loop. Instead, calculate all the required style changes first and then apply them together.

  - **Use CSS Transitions and Animations:** When animating or transitioning elements, prefer CSS transitions and animations over JavaScript-based approaches. CSS-based animations leverage hardware acceleration and are generally more efficient. They can be achieved using CSS classes and properties like transition and transform.

  - **Debounce and Throttle Event Handlers:** When attaching event handlers that perform expensive operations, consider debouncing or throttling the handlers to limit their execution frequency. Debouncing delays the execution of the handler until a certain time has passed without the event being triggered again. Throttling limits the execution frequency to a specific interval. These techniques can help optimize performance when dealing with events such as scrolling or resizing.

  - **Use Event Delegation:** Event delegation allows you to handle events on a parent element instead of attaching individual event handlers to multiple child elements. This can reduce the number of event listeners and improve performance, especially when dealing with dynamically generated or large numbers of elements.

  - **Optimize CSS Selectors:** When using CSS selectors to access elements, use the most specific selectors possible. Avoid overly generic selectors that can match a large number of elements. The browser's selector engine will need to spend more time searching for elements that match broad selectors, which can impact

performance.

- **Consider Virtual DOM Libraries:** In complex web applications with frequent DOM updates, consider using Virtual DOM libraries like React or Vue.js. These libraries handle efficient updates by maintaining a virtual representation of the DOM and performing targeted updates only to the necessary elements.

- By following these performance and optimization considerations, you can improve the efficiency of your JavaScript code when working with the DOM, resulting in faster rendering and a smoother user experience.
- **Note:** To improve the efficiency of your JavaScript code, you can reduce DOM access. Minimize the number of DOM queries and manipulations by caching references to frequently accessed elements. Store references in variables and reuse them instead of querying the DOM repeatedly.