**Batch: T7**

**Practical No: 4**

**Title of Assignment: Study of JavaScript and DOM**

**Student Name: Shreeyash Shripad Dongarkar**

**Student PRN: 22510025**

**Problem Statement 0: Basics of DOM**

**1) What is the DOM?**

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a web page as a tree of objects, where each node in the tree corresponds to a part of the document (such as an element, attribute, or piece of text). The DOM allows programs and scripts (like JavaScript) to dynamically access and update the content, structure, and style of a web page.

**Key Points about the DOM:**

* **Tree Structure**: The DOM organizes a document's structure as a hierarchical tree. The root of this tree is the document object, and the branches are the elements (e.g., <html>, <body>, <div>, etc.).
* **Nodes**: Each item in the DOM tree is called a node. Nodes can be elements, text, or comments.
* **Interactive**: The DOM allows developers to modify the document's content and structure in response to user actions or other events.
* **Language-Independent**: The DOM can be accessed and manipulated using various programming languages, though JavaScript is the most commonly used language for this purpose in web development.

**2) What is DOM Tree Structure? Elaborate its elements with help of example.**

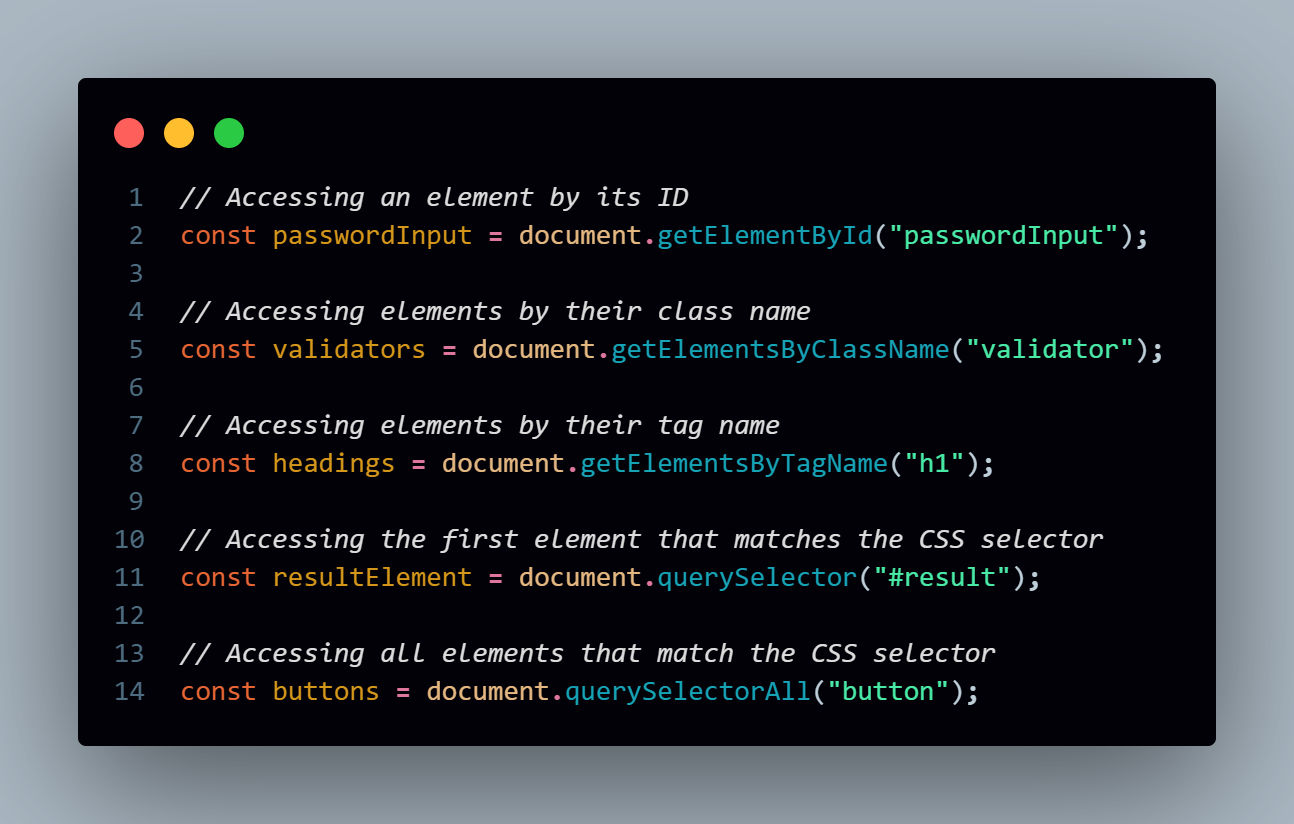
The DOM (Document Object Model) tree structure represents the HTML document as a tree of nodes, where each node corresponds to a part of the document. This structure allows you to access and manipulate the document’s content, structure, and style using scripting languages like JavaScript.

This can be explained using the example of DOM for Password Validator in Assignment 3 (Problem Statement 2)

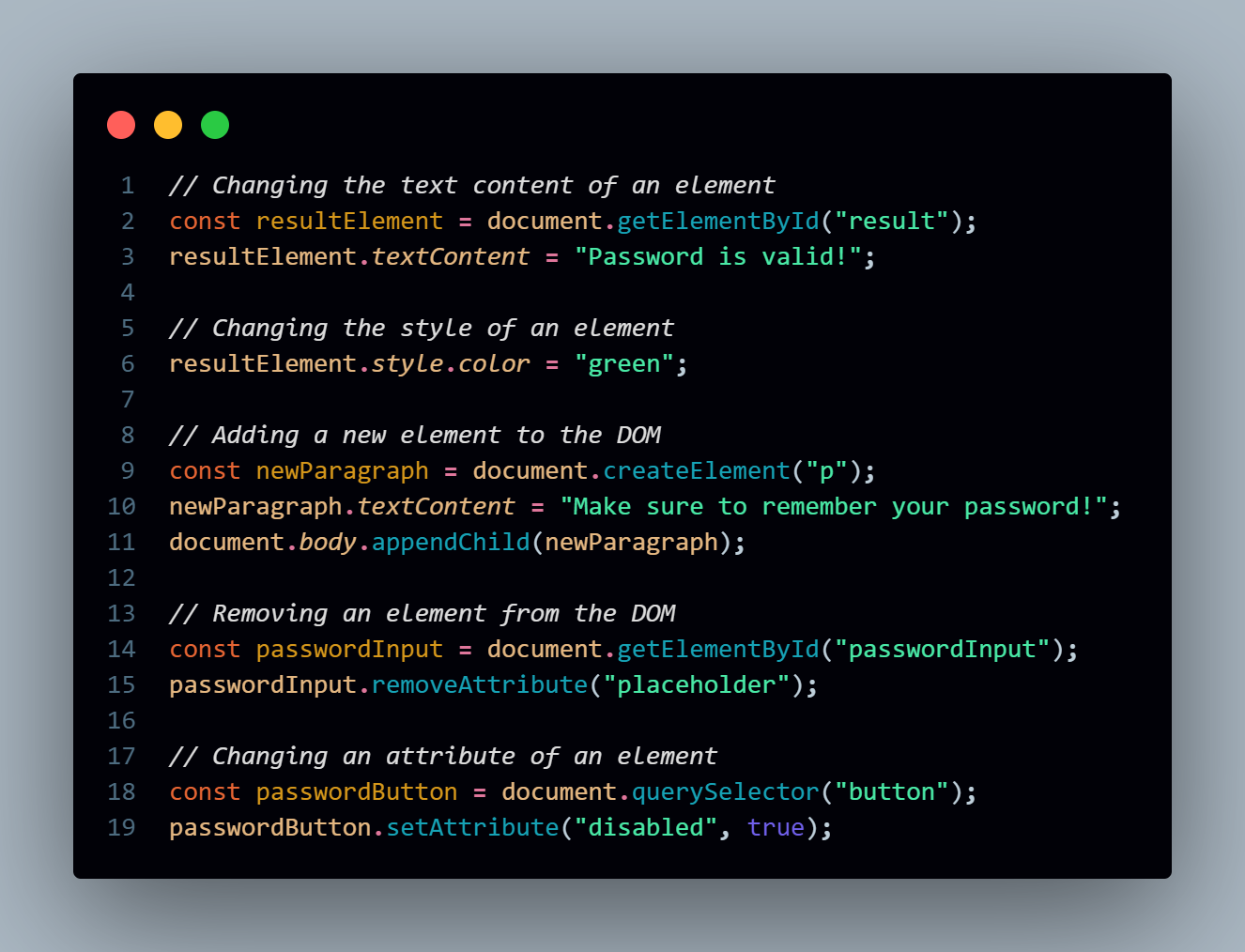


**3) Give Examples for:**

i) Accessing the DOM



ii) Manipulating the DOM



iii) Event Handling



iv) Traversing the DOM



**4) What are performance considerations while implementing the DOM and can DOM support all browsers?**

**Performance Considerations While Implementing the DOM**

When working with the DOM, it's important to consider performance, especially for complex or dynamic web applications. Here are some key performance considerations:

1. **Minimize DOM Manipulations**:
   * **Batch DOM Updates**: Frequent DOM manipulations can cause the browser to re-render the page multiple times, leading to performance bottlenecks. It's more efficient to batch updates together, making multiple changes at once.
   * **Use Document Fragments**: If you need to add several elements to the DOM, consider using a DocumentFragment to minimize reflows and repaints. You can append multiple elements to the fragment first, and then add the fragment to the DOM in one operation.
2. **Avoid Forced Synchronous Layouts**:
   * **Reading and Writing DOM Separately**: Avoid reading layout properties (like offsetHeight, clientHeight, etc.) immediately after making changes to the DOM. This can force the browser to re-calculate the layout, leading to performance issues.
3. **Use Efficient Event Handling**:
   * **Event Delegation**: If you need to handle events for many similar elements, use event delegation. Attach a single event listener to a parent element instead of adding individual listeners to each child element.
4. **Optimize Reflows and Repaints**:
   * **Minimize Layout Thrashing**: Avoid repeatedly changing the DOM and querying layout properties in the same function. This causes the browser to constantly reflow and repaint, which can slow down the page.
   * **CSS Changes**: Changes to certain CSS properties (like width, height, margin, padding) can trigger reflows, while others (like color, background-color) only trigger repaints. Optimize CSS to minimize reflows.
5. **Use requestAnimationFrame for Animations**:
   * **Smooth Animations**: For animations or visual updates, use requestAnimationFrame instead of setTimeout or setInterval. This method tells the browser to execute the callback before the next repaint, ensuring smoother animations and reducing CPU usage.
6. **Lazy Load Content**:
   * **Lazy Loading**: For large images or data-heavy content, consider lazy loading. This means loading content only when it's needed, such as when it comes into the viewport, to reduce initial load times.
7. **Use Virtual DOM for Complex Applications**:
   * **React and Virtual DOM**: In complex applications, using a framework like React with its virtual DOM can significantly improve performance. The virtual DOM minimizes direct DOM manipulations by updating only the parts of the DOM that have changed.

**Browser Support for DOM**

The DOM is a web standard, and all modern browsers support the core features of the DOM, including popular browsers like:

* **Google Chrome**
* **Mozilla Firefox**
* **Safari**
* **Microsoft Edge**
* **Opera**

**Considerations for Browser Compatibility:**

1. **Older Browsers**:
   * While modern browsers fully support the DOM, older browsers (like Internet Explorer 8 and below) may not support newer DOM methods and properties. Developers need to account for this if supporting older browsers.
   * For example, methods like querySelector and addEventListener are not supported in very old browsers.
2. **Polyfills**:
   * For compatibility with older browsers, developers can use polyfills—scripts that replicate the functionality of modern features in older browsers.
3. **Cross-Browser Testing**:
   * Always test your web application across different browsers to ensure consistent behavior. Browser-specific quirks or differences in implementation can sometimes cause issues.

**5) Elaborate common methods and properties of DOM**

**Common DOM Methods**

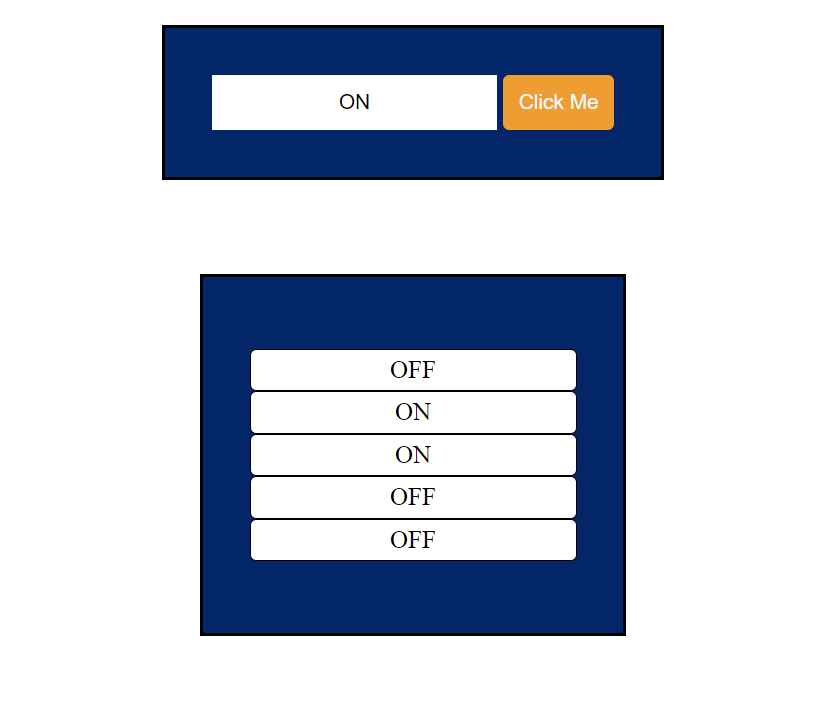
1. **Selecting Elements**
   * **getElementById(id)**: Returns the element with the specified ID. The ID should be unique within the document.
   * **getElementsByClassName(className)**: Returns a live HTMLCollection of elements with the specified class name.
   * **getElementsByTagName(tagName)**: Returns a live HTMLCollection of elements with the specified tag name.
   * **querySelector(selector)**: Returns the first element that matches the CSS selector. Supports more complex selectors.
   * **querySelectorAll(selector)**: Returns a static NodeList of elements that match the CSS selector.
2. **Creating and Modifying Elements**
   * **createElement(tagName)**: Creates a new element with the specified tag name.
   * **createTextNode(text)**: Creates a new text node with the specified text content.
   * **appendChild(child)**: Adds a child node to the end of the list of children of a specified parent node.
   * **insertBefore(newNode, referenceNode)**: Inserts a new child node before a specified existing child node.
   * **removeChild(child)**: Removes a specified child node from the DOM.
   * **replaceChild(newChild, oldChild)**: Replaces an old child node with a new node.
3. **Element Attributes**
   * **getAttribute(attributeName)**: Retrieves the value of a specified attribute on an element.
   * **setAttribute(attributeName, value)**: Sets the value of a specified attribute on an element.
   * **removeAttribute(attributeName)**: Removes a specified attribute from an element.
4. **Event Handling**
   * **addEventListener(eventType, callback)**: Adds an event listener to an element, allowing it to respond to events like clicks or key presses.
   * **removeEventListener(eventType, callback)**: Removes an event listener from an element.
5. **Node Traversal**
   * **parentNode**: Returns the parent node of the specified node.
   * **childNodes**: Returns a live NodeList of a node’s child nodes.
   * **firstChild**: Returns the first child node of the specified node.
   * **lastChild**: Returns the last child node of the specified node.
   * **nextSibling**: Returns the node immediately following the specified node.
   * **previousSibling**: Returns the node immediately preceding the specified node.

**Common DOM Properties**

1. **Element Properties**
   * **innerHTML**: Gets or sets the HTML content of an element.
   * **textContent**: Gets or sets the text content of an element, ignoring HTML tags.
   * **className**: Gets or sets the class attribute of an element.
   * **id**: Gets or sets the ID attribute of an element.
   * **style**: Provides access to the inline styles of an element.
   * **value**: Gets or sets the value of form elements, such as <input> or <textarea>.
2. **Document Properties**
   * **document.title**: Gets or sets the title of the document (the content inside the <title> tag).
   * **document.body**: Returns the <body> element of the document.
   * **document.documentElement**: Returns the root element of the document (typically <html>).
   * **document.forms**: Returns a collection of all forms in the document.
   * **document.images**: Returns a collection of all <img> elements in the document.

**Problem Statement 1: DOM Selector Methods**

Here I’ve used event handling to change the value of change the value of input element and list items based on given event

****

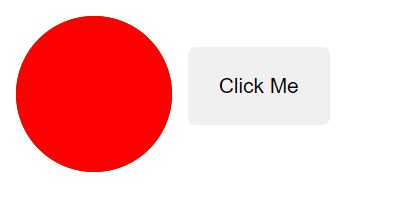
**Problem Statement 2: Events and User Interactions**

Here I’ve added eventListener to handle various events like click, mouseeneter, mouseleave to perform given changes



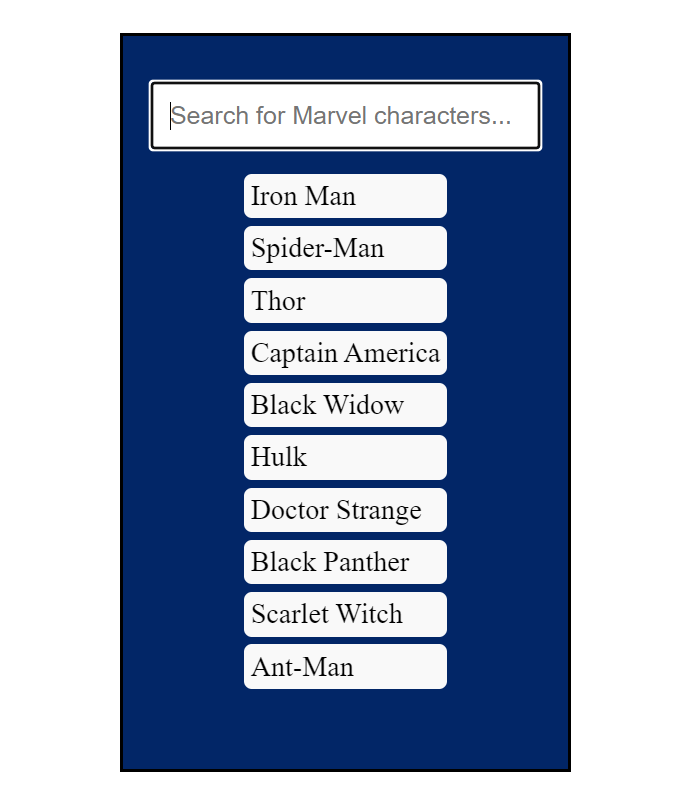
**Problem Statement 1: Dom Manipulation with Javascript**

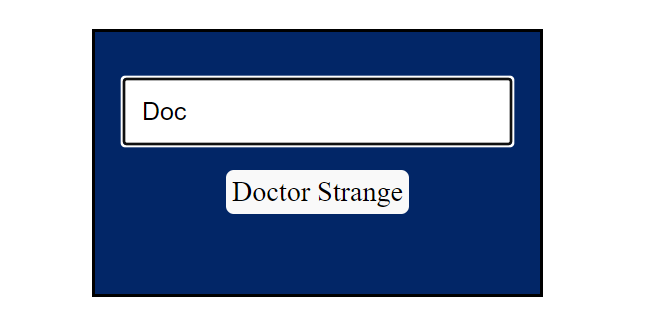
Here I’ve used eventListener for handling click event which will remove element from DOM

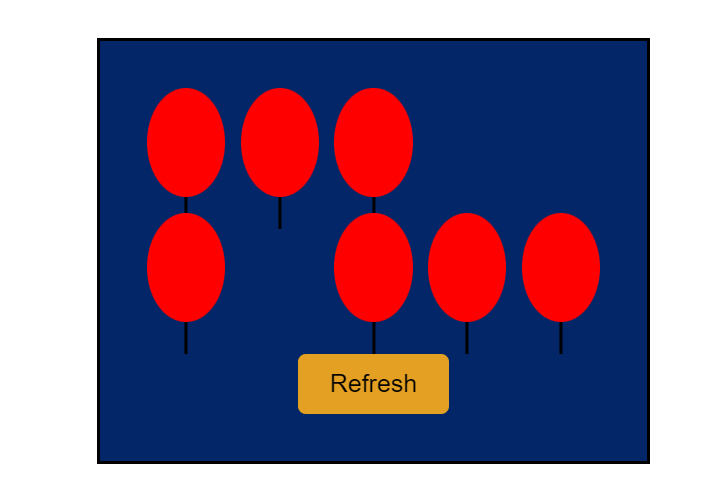
 

**Problem Statement 4:**









**Problem Statement 5: Recursive Functions**



