Document Object Model

Is a programming Interface for HTML and XML documents

It defines the logical structure of documents as a tree like represenetation using series of nodes and objects.

Programmers cancreate and build documents navigate their structure and add modify or delete elements and content.

Java script interprets DOM easily . It cant understand the tags in html. But can understand h1 in DOM

Javascritp can access each of the objects using difficult functions

querySelectorAll() method.

Essential DOM methods are

getElementId

getElementByTagName

Mode

createElement

appendChild

appendChild

getAttribute

setAttribute

innerHTML

document.forms

DOM is responsible for rendering ,updating and handling user interactions

Minimize DOM Access ? use caching to find a element.

Use document fragments for adding list of elements – in turn adds performance

document.createDocumentFragment()

use appendchild to indert into DOM

// Example of using a document fragment  
const fragment = document.createDocumentFragment();  
  
for (let i = 0; i < 1000; i++) { //create 1000 elements without affecting the DOM!  
 const element = document.createElement('div');  
 element.textContent = 'Item ' + i;  
 fragment.appendChild(element);  
}  
  
document.getElementById('container').appendChild(fragment);

Batch update

// Create a document fragment  
const fragment = document.createDocumentFragment();  
  
// Batch updates within the fragment  
const element1 = document.createElement('div');  
element1.textContent = 'Hello';  
element1.setAttribute('class', 'greeting');  
element1.style.color = 'blue';  
fragment.appendChild(element1);  
  
const element2 = document.createElement('div');  
element2.textContent = 'World';  
element2.setAttribute('class', 'greeting');  
element2.style.fontSize = '20px';  
fragment.appendChild(element2);  
  
const element3 = document.createElement('div');  
element3.textContent = '!';  
element3.setAttribute('class', 'greeting');  
element3.style.fontWeight = 'bold';  
fragment.appendChild(element3);  
  
// Append the fragment to the DOM  
const container = document.getElementById('container');  
container.appendChild(fragment);

Use event delegation

So, here’s how it works: we identify a common parent element, often referred to as the event delegation container. This container captures events that bubble up from its descendant elements. Instead of attaching event listeners to each descendant element separately, we delegate the responsibility of handling events to the container.

// Bad practice: Attaching event listeners to individual elements  
const buttons = document.querySelectorAll('.my-button');  
buttons.forEach(button => {  
 button.addEventListener('click', () => {  
 // Handle button click  
 });  
});  
  
// Good practice: Utilizing event delegation  
document.getElementById('container').addEventListener('click', event => {  
 if (event.target.matches('.my-button')) {  
 // Handle button click  
 }  
});

1. First and foremost, we reduce the number of event listeners attached to individual elements, resulting in improved performance and memory usage.
2. Secondly, event delegation allows us to handle dynamically added or removed elements effortlessly. Since we attach the event listener to a higher-level container, new elements that match the event criteria will automatically be covered. ç
3. Lastly, event delegation encourages cleaner and more maintainable code. With fewer event listeners scattered throughout the codebase, it’s easier to understand and modify the event handling logic.

//HTML  
<div id="container">  
 <button class="item">Button 1</button>  
 <button class="item">Button 2</button>  
 <button class="item">Button 3</button>  
</div>  
  
//JavaScript  
// Event delegation using a common parent element  
const container = document.getElementById('container');  
  
// Event handler function  
function handleClick(event) {  
 if (event.target.classList.contains('item')) {  
 // Handle the event for the specific element  
 console.log('Button clicked:', event.target.textContent);  
 }  
}  
  
// Attach event listener to the container  
container.addEventListener('click', handleClick);

We have 3 butttons but only a single handleClick function being called. Inside the function we can determine how we handle each execution of the handler based on the properties of the event’s target element.

With event delegation in your arsenal, you’ll gracefully handle events with efficiency and elegance, leaving your codebase lighter, more adaptable, and easier to maintain.

Debounce or event throttler

Debouncing delays the execution of an event handler until a certain period of inactivity has passed.

On the other hand, throttling limits the rate at which the event handler is invoked, ensuring it’s executed at a controlled interval.

* **Debouncing**: When using debouncing, the event handler will only execute once after a specified delay since the last event trigger. This is useful when you want to respond to the final state of an event after a period of inactivity — like doing a live search when the user is typing, you want the search to happen when they enter the last character, **not on every character**— .

function debounce(func, delay) {  
 let timeoutId;  
   
 return function(...args) {  
 clearTimeout(timeoutId);  
 timeoutId = setTimeout(() => {  
 func.apply(this, args);  
 }, delay);  
 };  
}  
  
// Usage example  
const debouncedEventHandler = debounce((event) => {  
 // Your event handling logic here  
}, 300);  
   
window.addEventListener('scroll', debouncedEventHandler);

In the above code snippet, the debounce function takes an event handler function func and a delay delay as parameters. It returns a debounced version of the event handler that will execute after the specified delay since the last event trigger. You can then attach this debounced event handler to the desired event listener, like scroll.

* **Throttling**: With throttling, the event handler will be executed at a controlled interval, ensuring it’s not invoked more frequently than the specified delay.

function throttle(func, delay) {  
 let isThrottled = false;  
   
 return function(...args) {  
 if (!isThrottled) {  
 func.apply(this, args);  
 isThrottled = true;  
 setTimeout(() => {  
 isThrottled = false;  
 }, delay);  
 }  
 };  
}  
  
// Usage example  
const throttledEventHandler = throttle((event) => {  
 // Your event handling logic here  
}, 300);  
  
window.addEventListener('resize', throttledEventHandler);

In this code snippet, the throttle function takes an event handler function func and a delay delay as parameters. It returns a throttled version of the event handler that will execute at a controlled interval, ensuring it's not invoked more frequently than the specified delay. The throttled event handler can be attached to the desired event listener, like resize.

By applying debouncing or throttling techniques, we gain control over the execution frequency of our event handlers, preventing performance issues caused by excessive event triggers.