2025 Volume: 01 Issue: 07 www.ijrpa.com ISSN 2456-9995 Review Article

International Journal Research Publication Analysis

Page: 01-11

"DOM MANIPULATION AND EVENT HANDLING IN JAVASCRIPT"

***Aditya Raj Thakur, Dr. Vishal Shrivastava, Dr. Akhil Pandey**

Computer Science & Engineering, Arya College of Engineering & I.T. Jaipur, India.

Article Received: 30 October 2025
Article Revised: 19 November 2025
Published on: 10 December 2025

***Corresponding Author: Aditya Raj Thakur**
Computer Science & Engineering, Arya College of Engineering & I.T. Jaipur, India. DOI: https://doi-doi.org/101555/ijrpa.4845

ABSTRACT

Document Object Model (DOM) manipulation and event handling are fundamental concepts in JavaScript that enable developers to create dynamic, interactive, and user-friendly web applications. The DOM represents the structure of a webpage as a hierarchical tree, where each element can be accessed, modified, added, or removed using JavaScript. developers can improve performance, usability, and maintainability of web applications while ensuring seamless interaction between users and systems.

INTRODUCTION

One of the most unique and useful abilities of JavaScript is its ability to manipulate the DOM. But what is the DOM, and how do we go about changing it? Let‟s jup right.

Lesson overview

This section contains a general overview
of topics that you will learn in this lesson.

- Explain what the DOM is in relation to a webpage.
- Explain the difference between a "node" and an "element".
- Explain how to target nodes with "selectors".
- Explain the basic methods for finding, adding, removing, and altering DOM nodes.
- Explain the difference between a "NodeList" and an "array of nodes".
- Explain what "bubbling" is and how it works.

Document Object Model

The DOM (or Document Object Model) is a tree-like representation of the contents of a webpage - a tree of "nodes" with different relationships depending on how they‟re arranged in the HTML document. There are many types of nodes, most of which are not commonly used. In this lesson we will be focusing on "element" nodes which are primarily used for manipulating the DOM.

```
<div id="container">
<div class="display"></div>
<div class="controls"></div>
</div>
```

In the above example, the <div class="display"></div> is a "child" of <div id="container"></div> and a "sibling" to <div class="controls"></div>. Think of it like a family tree. <div id="container"></div> is
a **parent**, with its **children** on the next
level, each on their own "branch".

Targeting nodes with selectors When working with the DOM, you use "selectors" to target the nodes you want to work with. You can use a combination of CSS-style selectors and relationship properties to target the nodes you want. Let"s start with CSS-style selectors. In the above example,you could use the following selectors to referto <div class="display"></div>:

- div.display
- .display
- #container > .display
- div#container > div.display

You can also use relational selectors (i.e., firstElementChild or lastElementChild, etc.) with special properties owned by the nodes.

```
// selects the #container div (don't worry about the syntax, we'll get there)
const                    container                   = document.querySelector("#container");

// selects the first child of #container => .display const display = container.firstElementChild;
    console.log(display);              //    <div
    class="display"></div>
    // selects the .controls div
    const                controls          =
    document.querySelector(".contr
```

ols");

// selects the prior sibling => .display
const display = controls.previousElementSibling; console.log(display); // <div class="display"></div>

So you‟re identifying a certain node based
on its relationships to the nodes around it.

DOM methods

When your HTML code is parsed by a web browser, it is converted to the DOM, as was mentioned above. One of theprimary differences is that these nodes are JavaScript objects that have many properties and methods attached to them. These properties and methods are the primary tools we are going to use to manipulate our webpage with JavaScript.

Query selectors

- element.querySelector(selector) - returns a reference to the first match of selector.
- element.querySelectorAll(selectors) - returns a "NodeList" containing references to all of the matches of the selectors.

Performance consideration

There are several other, more specific queries, that offer potential (marginal) performance benefits, but we won‟t be going over them now.

It‟s important to remember that when using querySelectorAll, the return value is **not** an array. It looks like an array, and it somewhat acts like an array, but it‟s really a "NodeList". The big distinction is that several array methods are missing from NodeLists. One solution, if problems arise, is to convert the NodeList into an array. You can do this with Array.from() or the spread operator.

Element creation

- document.createElement(tagName, [options]) - creates a new element of tag type tagName. [options] in this case means you can add some optional parameters to the function. Don‟t worry about these at this point.

const div = document.createElement("div");

This function does NOT put your new element into the DOM - it creates it in memory. This is so that you can manipulate the element (by adding styles, classes, ids, text, etc.) before placing it on the page. You can place the element into the DOM with one of the following methods.

Append elements

- parentNode.appendChild(childNode) -appends childNode as the last child of parentNode.
- parentNode.insertBefore(newNode, referenceNode) -inserts newNode into parentNode before re ferenceNode.

  Remove elements
- parentNode.removeChild(child) - removes child from parentNode on the DOM and returns a reference to child. Altering elements

When you have a reference to an element, you can use that reference to alter the element‟s own properties. This allows you to do many useful alterations, like adding, removing, or altering attributes, changing classes, adding inline style information, and more.

// creates a new div referenced in the variable 'div' const div = document.createElement("div"); Adding inline style
// adds the indicated style rule to the element in the div variable
div.style.color = "blue";

// adds several style rules
div.style.cssText = "color: blue; background: white;";

// adds several style rules
div.setAttribute("style", "color: blue; background: white;");
When accessing a kebab-cased CSS property like background-color with JS, you will need to either use camelCase with dot notation or bracket notation. When using bracket notation, you can use either camelCase or kebab-case, but the property name must be a string.
// dot notation with kebab case: doesn't work as it attempts to subtract color from div.style.background
// equivalent to: div.style.background - color div.style.background-color;

// dot notation with camelCase: works, accesses the div's background-color style div.style.backgroundColor;

// bracket notation with kebab-case: also works div.style["background-color"];

// bracket notation with camelCase: also works div.style["backgroundColor"];

Editing attributes

// if id exists, update it to 'theDiv', else create an id with value "theDiv"

div.setAttribute("id", "theDiv");

// returns value of specified attribute, in this case "theDiv"

div.getAttribute("id");

// removes specified attribute div.removeAttribute("id");

See MDN"s section on HTML Attributes for more information on available attributes.

Working with classes

// adds class "new" to your new div div.classList.add("new");

// removes "new" class from div div.classList.remove("new");

// if div doesn't have class "active" then add it, or if it does, then remove it div.classList.toggle("active");

It is often standard (and cleaner) to toggle a CSS style rather than adding and removing inline CSS.

Adding text content

// creates a text node containing "Hello World!" and inserts it in div

div.textContent = "Hello World!";

Adding HTML content

// renders the HTML inside div

div.innerHTML = "<span>Hello World!</span>";

Security risks of innerHTML

Using textContent is preferred over innerHTML for adding text, as innerHTML should be used sparingly to avoid potential security risks. To understand the dangers of using innerHTML, watch this video about preventing the most common cross-site scripting attack.

Let‟s take a minute to review what we‟ve covered and give you a chance to practice this stuff before moving on. Check out this example of creating and appending a DOM element to a webpage.

```
<!-- your HTML file: -->
<body>
<h1>THE TITLE OF YOUR WEBPAGE</h1>
<div id="container"></div>
</body>
// your JavaScript file
constcontainer= document.querySelector("#container");

const content = document.createElement("div"); content.classList.add("content");
content.textContent = "This is the glorious text- content!";
container.appendChild(content);
```

In the JavaScript file, first we get a reference to the container div that already exists in our HTML. Then we create a new div and store it in the variable content. We add a class and some text to the content div and finally append that div to container. After the JavaScript code is run, our DOM tree will look like this:

```
<!-- The DOM -->
<body>
<h1>THE TITLE OF YOUR WEBPAGE</h1>
<div id="container">
  <div class="content">This is the glorious text- content!</div>
</div>
</body>
```

Keep in mind that the JavaScript does not alter your HTML, but the DOM -

your HTML file will look the same, but the JavaScript changes what the browser renders.
Timing of JavaScript

Your JavaScript, for the most part, is run whenever the JS file is run or when the script tag is encountered in the HTML. If you are including your JavaScript at the top of your file, many of these DOM manipulation methods will not work because the JS code is being run before the nodes are created in the DOM. The simplest way to fix this is to include your JavaScript at the bottom of your HTML file so that it gets run after the DOM nodes are parsed and created.
Alternatively, you can link the JavaScript file in the <head> of your HTML document. Use the <script> tag with the src attribute containing the path to the JS file, and include the defer keyword to load the file after the HTML is parsed, as such:

```
<head>
<script src="js-file.js" defer></script>
</head>
```

Find out more about the defer attribute for script tags.

Exercise

Copy the example above into files on your own computer. To make it work, you"ll need to supply the rest of the HTML skeleton and either link your JavaScript file or put the JavaScript into a script tag on the page. Make sure everything is working before moving on!

Add the following elements to the container using ONLY JavaScript and the DOM methods shown above:

1. a <p> with red text that says "Hey I"m red!"
2. an <h3> with blue text that says "I"m a blue h3!"
3. a <div> with a black border and pink background color with the following elements inside of it:

- another <h1> that says "I"m in a div"
- a <p> that says "ME TOO!"
- Hint for this one: after creating the <div> with createElement, append the <h1> and <p> to it before adding it to the container.

Events

Now that we have a handle on manipulating the DOM with JavaScript, the next step is learning how to make that happen dynamically or on demand! Events are how you make that magic happen on your pages. Events are actions that occur on your webpage, such as mouse-clicks or key-presses. Using JavaScript, we can make our webpage listen to and react to these events.

There are three primary ways to go about this:

- You can specify function attributes directly on your HTML elements.
- You can set properties in the form of on<eventType>, such as onclick or onmousedown, on the DOM nodes in your JavaScript.
- You can attach event listeners to the DOM nodes in your JavaScript.

Event listeners are definitely the preferred method, but you will regularly see the others in use, so we"re going to cover all three.

We"re going to create three buttons that all alert "Hello World" when clicked. Try them all out using your own HTML file or
using something like CodePen. Method 1
<button onclick="alert('HelloWorld')">Click Me</button>

This solution is less than ideal because we"re cluttering our HTML with JavaScript. Also, we can only set one "onclick" property per DOM element, so we"re unable to run multiple separate functions in response to a click event using this method.

Method 2
<!-- the HTML file -->
<button id="btn">Click Me</button>
// the JavaScript file
const btn = document.querySelector("#btn"); btn.onclick = () => alert("Hello World");
Reviewing arrow function syntax
If you need to review the arrow syntax ()
=>, check this article about arrow functions.

This is a little better. We"ve moved the JS out of the HTML and into a JS file, but we still have the problem that a DOM element can only have one "onclick" property.

Method 3

```
<!-- the HTML file -->
<button id="btn">Click Me Too</button>
// the JavaScript file
const btn = document.querySelector("#btn"); btn.addEventListener("click", () => {
alert("Hello World");
});
```

Now, we maintain separation of concerns, and we also allow multiple event listeners if the need arises. Method 3 is much more flexible and powerful, though it is a bit more complex to set up.

Note that all three of these methods can be used with named functions like so:

```
<!-- the HTML file -->
<!-- METHOD 1 -->
<button onclick="alertFunction()">CLICK ME BABY</button>
// the JavaScript file
// METHOD 1
function alertFunction() { alert("YAY! YOU DID IT!");
}
<!-- the HTML file -->
<!-- METHODS 2 & 3 -->
<button id="btn">CLICK ME BABY</button>
// the JavaScript file
// METHODS 2 & 3
function alertFunction() { alert("YAY! YOU DID IT!");
```

the **event** itself. Within that object you have access to many useful properties and methods (functions that live inside an object) such as which mouse button or key was pressed, or information about the event‟s **target** - the DOM node that was clicked. There‟s nothing magical about e as a name or where it comes from. JavaScript knows the parameter is an event because an event listener callback takes an Event object by definition. When the callback is run, the event handler passes in its own reference to the event. You can read more about the event objects on MDN‟s introduction to events.

```
}
const btn = document.querySelector("#btn");
// METHOD 2
btn.onclick = alertFunction;
// METHOD 3
```

btn.addEventListener("click", alertFunction); Using named functions can clean up your code considerably, and is a really good idea if the function is something that you are going to want to do in multiple places.

With all three methods, we can access more information about the event by passing a parameter to the function that we are calling. Try this out on your own machine:

```
btn.addEventListener("click", function (e) { console.log(e);
});
```

Understanding callbacks

When we pass in alertFunction or function (e)

{...} as an argument to addEventListener, we call this a callback. A callback is simply a function that is passed into another function as an argument.

The e parameter in that callback function contains an object that references Try this:

```
btn.addEventListener("click", function (e) { console.log(e.target);
});
```

and now this:

```
btn.addEventListener("click", function (e) { e.target.style.background = "blue";
});
```

Pretty cool, eh?

Attaching listeners to groups of nodes

This might seem like a lot of code if you‟re attaching lots of similar event listeners to many elements. There are a few ways to go about doing that more efficiently. We learned above that we can get a NodeList of all of the items matching a specific selector with querySelectorAll('selector'). In order to add a listener to each of them, we need to iterate through the whole list, like so:

```
<div id="container">
<button id="one">Click Me</button>
<button id="two">Click Me</button>
<button id="three">Click Me</button>
</div>
// buttons is a node list. It looks and acts much like an array.
constbuttons= document.querySelectorAll("button");

// we use the .forEach method to iterate through each button
buttons.forEach((button) => {
// and for each one we add a 'click' listener button.addEventListener("click", () => {
alert(button.id);
});
});
```

This is just the tip of the iceberg when it comes to DOM manipulation and event handling, but it‟s enough to get you started with some exercises. In our examples so far, we have been using the „click‟ event exclusively, but there are many more available to you.

References

1. Flanagan, D. (2020). JavaScript: The Definitive Guide. O‟Reilly Media.
2. Crockford, D. (2008). JavaScript: The Good Parts. O‟Reilly Media.
3. Mozilla Developer Network (MDN). “DOM Manipulation.” Available at: https://developer.mozilla.org/
4. Mozilla Developer Network (MDN). “Event Handling in JavaScript.” Available at: https://developer.mozilla.org/
5. W3Schools. “JavaScript HTML DOM.” Available at: https://www.w3schools.com/js/js_htmldom.asp
6. Duckett, J. (2014). JavaScript and jQuery: Interactive Front-End Web Development. Wiley.