**DOM and Events**

**What is Document Object Model DOM?**

DOM is a cross platform and language independent interface that treats an HTML/XML/SVG document as a tree structure wherein each node is an object representing a part of the document (or you can say each node represents an HTML element).

This interface allows us to create, change or remove elements from the document. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page. Nodes can also have eventhandlers attached to them. Once an event is triggered, the event handler gets executed.

When a web browser parses an HTML document, it builds a DOM tree and then uses it to display the document.

Take a look at this HTML code:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>DOM tree structure</title>

</head>

<body>

<h1>DOM tree structure</h1>

<h2>Learn about the DOM</h2>

</body>

</html>

Our document is called the root node and contains one child node which is the <html> element. The <html> element contains 2 children which are <head> and <body> element.

DOM methods allow programmatic access to the tree (using JS language).

**Fundamental Data Types/Interfaces**

- **Document** : the root document object itself.

- **Node** : Every object located within a document is a node of some kind. In HTML document,an object can be an element node, text node or attr node etc.

- **Element** : The *element* type is based on a *node*. It refers to an element or a node of type *element* returned by a member of DOM API.

- **NodeList** : It is an array of elements, like the kind that is returned by the method document.querySelectorAll().

- **Attr** : When an attribute is returned by a member, it is an object reference that exposes a small special interface for attributes.

**Relation between Objects and Interfaces in DOM**

In a DOM, one object implements either one interface or may be several interfaces. The table object, for e.g, implements a specialized HTMLTableElement interface, which includes such methods as CreateCaption and insertRow. But since it’s an HTML element ultimately, it also implements *Element* interface. And finally, since an HTML element is also, as far as the DOM is concerned, a node in the tree of nodes that make up the object model for an HTML page, the table also implements a more basic *Node* interface, from which *Element* derives.

The document and window objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, the window object represents something like the browser, and the document object is the root of the document itself. Element inherits from the generic Node interface, and together these two interfaces provide many of the methods and properties you use on individual elements.

Following is the list of commonly used APIs:

document.getElementById()

document.querySelector()

document.querySelectorAll()

document.createElement()

Element.innerHTML

Element.setAttribute()

Element.getAttribute()

EventTarget.addEventListener()

**Events**

Events are the things that happen in the system you are programming, which the system tells you about so your code can react to them. The system produces or *fires* a signal of some kind when an event occurs, and provides a mechanism by which an action can be taken automatically. Events are fired inside the browser window, and tend to be attached to a specific item that resides in it. This might be a single element, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events which can occur:

* The user selects, clicks, or hovers the cursor over a certain element.
* The user chooses a key on the keyboard.
* The user resizes or closes the browser window.
* A web page finishes loading.
* A form is submitted.
* A video is played, paused, or ends.
* An error occurs.

These are some events:

* click
* keyup
* keydown
* mousehover
* submit

Every event fired has a corresponding event object associated with it. Like keydown/keyup’s event object is KeyBoard event, which is a specialized Event object with a key property that tells you which key was pressed.

To react to an event, you attach an event handler to it. This is a block of code (usually a JS function that you as a programmer create) that runs when the event fires. When such a block of code is defined to run in response to an event, we say we are *registering an event handler*.

Note: Event handlers are sometimes called event listeners – they are pretty much interchangeable, although strictly speaking, they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

You can add multiple eventlisteners to an event and all will get executed.

Example:

HTML:

<button>Change color</button>

JS:

const btn = document.querySelector("button");

function random(number) {

return Math.floor(Math.random() \* (number + 1));

}

btn.addEventListener("click", () => {

const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;

document.body.style.backgroundColor = rndCol;

});

It will set randomly generated colour for the button.

There are several other ways for registering the event handler. The example you saw above is the recommended way of doing it though, as it’s more powerful and scales best in complex programs in JS.

2 other ways you may see are these:

* Event Handler Properties
* Inline event handlers

**Event Handler Properties**

Objects that can fire events usually have properties whose name is *on* followed by the name of the event. This is called an event handler property.

Eg:

const btn = document.querySelector("button");

function random(number) {

return Math.floor(Math.random() \* (number + 1));

}

btn.onclick = () => {

const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;

document.body.style.backgroundColor = rndCol;

};

However, here you can’t add more than one event handler for a single event. Any subsequent attempt to set the property will override the earlier ones.

**Inline Event Handler/ Event Handler HTML attributes – Never use these**

Eg:

HTML

<button onclick="bgChange()">Press me</button>

JS

function bgChange() {

const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;

document.body.style.backgroundColor = rndCol;

}

Not suggested to use these as they become difficult to manage and hard to maintain.

**Event Object**

Sometimes, inside an event handler function, you’ll see a parameter specified with a name such as *event, evt or e.* This is called the event object, and it is automatically passed to event handlers to provide extra features and information.

It always has a *target* property which is a reference to the object event occurred upon.

Note – you can name this event anything in the JS function, but its customary to name it event/evt/e

**Preventing Default Behavior**

Sometimes, you’ll come across situations where you want to prevent an event from doing what it does by default. The most common example is that of a web form, for e.g a customer registration form. When you fill in the details and click the submit button, the natural behavior is for the data to be submitted to the server for processing.

The trouble comes when the user has not submitted the form correctly – as a developer you want to prevent the submission to the server and give an error message. Some browsers supports automatic form validation features, but since many don’t, you are advised not to rely on those and implement your own validation checks.

You can call event.preventDefault() function for this.

**Event Propagation**

It is a blanket term which is used to refer to 2 concepts:

* Event Bubbling
* Event Capture

Consider this example:

<div>

    <ul>

        <li><a href="placeholder.com"><img src="example.com" alt=""></a></li>

        <li><a href="placeholder.com"><img src="example.com" alt=""></a></li>

        <li><a href="placeholder.com"><img src="example.com" alt=""></a></li>

    </ul>

</div>

By clicking on img tag, it will generate click event for not only the <img> element, but also for its parent element <a>, as well as its grandparent element <li> and so on. It will go all the way up until it terminates at the window object. So the flow of element is this:

Img -> a > li -> div -> body -> html -> document -> window

It can be divided into 3 phases:

* Capture phase – from window to event target parent
* Target phase – target itself
* Bubble phase – From the event target back up to the window

What differentiates between these phases are the types of event listeners that are called.

During the *event capture* phase, only the *capturer* event listeners are called. These are the event listeners that were registered with a value of true for the third parameter of .addEventListener().

Eg: div.addEventListener(‘click’, listener, true);

During the *event target* phase, all the event listeners registered on the event target are invoked. They are all invoked regardless of the value of their capture flag.

Lastly, during the *event bubbling* phase, all other event listeners are called that do not have the 3rd parameter of true in the addEventListener function. Any event listener will be called only once.

**Event Bubbling**

Events in HTML always bubble up from innermost element to the top.

So let’s say there is a <button> inside a <div> and both have click event listeners. Then on clicking the button, button’s listener will be called first and then div’s.

Let’s say on the click of button, you don’t want div’s click handler to be called. Event object in DOM has a function called stopPropagation which prevents the event from bubbling up to any other parent element.

**Event Capture**

An alternative form of event propagation is *event capture*. This is like the event bubbling, but in reverse order; here the event fires from top level to the innermost element.

Event capture is disabled by default, to enable it you have to pass the *capture* option in *addEventListener()*.

**Stop Propagation**

All event listeners registered on the node on the propagation path that follow the event target will not be called. All other remaining listeners attached to the event target will still receive the event.

Take a look at this example to understand:

<!DOCTYPE html>

<html>

    <head>

        <title>

            This is a webpage

        </title>

    </head>

    <body>

        <div id="divone">

            <h1 id="hone">My First Heading</h1>

            <p id="para">My first paragraph.</p>

        </div>

    </body>

</html>

<script>

    const divone = document.querySelector("#divone");

    const hone = document.querySelector("#hone");

    const para = document.querySelector("#para");

    divone.addEventListener('click', (event) => {

        event.stopPropagation();

        alert("div clicked");

    });

    hone.addEventListener('click', (event) => {

        alert("hone clicked");

    });

    para.addEventListener('click', (event) => {

        event.stopPropagation();

        alert("para clicked");

    });

    document.body.addEventListener('click', () => {

        alert("body clicked");

    });

    document.body.addEventListener('click', () => {

        alert("body clicked during capture");

    }, true);

</script>

**Why do we have both Bubble and Capture?**

In the bad old days, when browsers were much less cross-compatible than now, Netscape only used event capturing, and Internet Explorer used only event bubbling. When the W3C decided to try to standardize the behavior and reach a consensus, they ended up with this system that included both, which is what modern browsers implement.

**Event Delegation**

It is a process in which if we want to assign an event listener to multiple child elements of a single parent, we register the event with just the parent element and have events bubble up, rather than attaching eventlistener to every child element.

Note: addEventListener() exists in EventTarget interface which is implemented by objects that can receive events and may have listeners for them.