15 – Event Handling

Events are things that happen in the system you are programming, causing the system to trigger a signal of some kind. Events are fired inside the browser window and tend to be attached to a specific item that resides in it (clicking a button(s), hovering over an element, typing in an input field, etc.).

To react to an event, an event handler is attached to it. This is a block of code that runs when the event fires.

The terms event handlers and event listeners can be used interchangeably. However, there is a subtle distinction between the two. An event listener is a mechanism that waits for an event to occur. Whereas, an event handler is a block of code, typically a function, that is executed in response to an event occurring.

**addEventListener()**

Objects that can trigger events have an addEventListener() method, which is the recommended mechanism for adding event listeners.

Example:

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;

});

In the above scenario, the HTML <button> triggers an event. Hence, the method addEventListener is called, in which we’re taking two parameters:

1) The string “click” indicates that we want to listen to the click event.

2) The callback function to call when the event takes place.

**Important Event Listeners**click: When an element is clicked  
dblclick: When an element is double-clicked  
mousedown: When the mouse button is pressed  
mouseup: When the mouse button is released  
mouseenter: When the mouse enters the element (No bubbling)  
mouseleave: When the mouse leaves the element (No bubbling)  
mouseover: When the mouse moves over an element (Bubbles)

Event Bubbling – The event an element receives is bubbled (transmitted) to its parents and ancestors, upward in the DOM, until the root element. This transmission can be impeded by using the stopPropagation() property of the event object.

keydown: When a key is pressed (fires repeatedly if key is held)

keyup: When a key is released

submit: When a form is submitted

input: On every change to a form field

change: When the value of an input/textarea/select is changed and then loses focus

DOMContentLoaded: When the HTML is fully loaded and pasred (but the images/styles may not be).

load: When the whole page is loaded (including images/styles)

resize: When the window is resized

scroll: When the user scrolls the page

**Remove Event Listener**

If an event listener has been added using addEventListener(), then it can be removed using the removeEventListener() method.

**Adding Multiple Event Listeners**

The recommended way to add event handlers is by using the addEventListener method for multiple reasons, including how well it scales in complex code. Another major reason is that it allows adding multiple listeners for a single event.

By making more than one call to addEventListener() and providing different handlers, one can add multiple event handlers for the same event:

myElement.addEventListener("click", functionA);

myElement.addEventListener("click", functionB);

Both functionA and functionB would occur when the element is clicked.

**Other Event Handler Properties**

The recommended and best way to add event handlers is through the addEventListener method. However there are two other ways of registering event handlers:

1. Event Handler Properties
2. Inline Event Handlers

**Event Handler Properties**

Objects (such as buttons) that can trigger events usually have properties whose name is “on” followed by the name of the event. This is known as an Event Handler Property.

To listen for the event, you can assign the handler function to the property using the “=” operator.

Example:

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

function random(number) {

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

}

function bgChange() {

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

document.body.style.backgroundColor = rndCol;

}

btn.onclick = bgChange;

**Important Event Handler Properties**

onclick: Element is clicked  
onchange: Value of a form field changes and loses focus  
oninput: Value of input changes live while typing  
onmouseover: Mouse enters element  
onmouseout: Mouse pointer leaves element  
onkeydown: Any key is pressed  
onkeyup: Key is released  
onsubmit: Form is submitted  
onfocus: Element gains focus  
onblur: Element loses focus  
onload: Page is loaded  
onresize: Browser window is resized  
onscroll: Page is scrolled

With the event handler property, multiple handler functions cannot be assigned for the same event to the same element. If done, the subsequent event handler function will overwrite all previous ones.

element.onclick = function1;

element.onclick = function2; // Only function2 will execute

**Inline Event Handlers (Not to be Used)**

The oldest method of handling events involved event handling HTML attributes (inline event handlers).

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

JavaScript:  
function bgChange() {

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

document.body.style.backgroundColor = rndCol;

}

These are now considered bad practices. It is not a good idea to mix up the HTML code with JS. Keeping a separate JS file allows one to apply it to multiple HTML documents. Even for single documents, using the same handler for 100 buttons will be redundant and would turn into a maintenance nightmare.

With addEventListener in JavaScript, one can easily add an event handler function to all the buttons with the following code:

const buttons = document.querySelectorAll(“button”);

for (const button of buttons) {

button.addEventhandler(“click”, bgChange)

}

Lastly, many server configurations disallow HTML Event Handler attributes as a security measure. Hence, NEVER use Inline Event Handlers.

**Event Objects**

A parameter can be specified in the handler function, usually as “event” or “e”. This is called an event object, which is passed automatically to event handlers to provide additional functionalities and features on the event that occurred.

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

function random(number) {

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

}

function bgChange(e) {

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

e.target.style.backgroundColor = rndCol;

console.log(e);

}

btn.addEventListener("click", bgChange);

In the above code, we set the background color style of e.target, the button itself. The target property of the event object always refers to the element on which the event occurred.

**Capture v/s Bubble**

When an event occurs, it doesn’t just affect that very element on which it was initially called. Instead, event propagation follows a three-phase process through the DOM:

1) Capture Phase

The event starts at the top and travels down to the target element.

window 🡪 document 🡪 html 🡪 body 🡪 button

One can listen to events in this phase using:  
element.addEventListener("click", handler, true);

2) Target

The event reaches the exact element that was interacted with. Any event listeners directly on the target element will fire. This happens regardless of whether the handler is listening in capture or bubble mode.

3) Bubbling Phase (Default)

The event bubbles up from the target element back towards the root of the DOM.

button 🡪 body 🡪 html 🡪 document 🡪 window

Event listeners are by default in this phase (instead of capture):  
element.addEventListener("click", handler); // Defaults to bubble phase

**Important Properties of Event Handlers**

type: The type of event (“click”, “keydown”, etc.)  
target: The actual element on which the event occurred  
currentTarget: The element to which the event handler is attached (may differ from target due to bubbling).  
timeStamp: The time (in ms) the event was created.

clientX, clientY: Mouse position relative to the viewport  
pageX, pageY: Mouse position relative to the entire document

button: Indicates which button was pressed  
key: The actual character or key pressed (“a”, “1”)  
code: The physical key on the keyboard (“KeyA”, “ArrowUp”)  
repeat: TRUE if the key is being held down and fired repeatedly

target.value: The current value of an input, textarea, or select element  
target.checked: For checkboxes and radio buttons

bubbles: Indicates if the event bubbles up through the DOM  
cancelable: Indicates if the event’s default behavior can be prevented  
stopPropagation: Stops the event from bubbling further  
stopImmediatePropagation: Stops other listeners of the same event from being called  
preventDefault: Prevents the default behavior of an element

**Prevent Default Behavior**In certain situations, one might need to prevent the default behavior of an event from taking place. For example, on submitting a form, the natural behavior is for the data to be submitted to the server. However, if the data entered by the user is incorrect, you would want to prevent the default behavior from occurring. Some browsers support automatic data validation, but many don’t. Thus, we implement our own validation checks and prevent the data from being submitted to the server using preventDefault().

Example:

const form = document.querySelector("form");

const fname = document.getElementById("fname");

const lname = document.getElementById("lname");

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

form.addEventListener("submit", (e) => {

if (fname.value === "" || lname.value === "") {

e.preventDefault();

para.textContent = "You need to fill in both names!";

}

});