Event Listeners

Event listeners are among the most frequently used JavaScript structures in web design. They allow us to add interactive functionality to HTML elements by “listening” to different events that take place on the page, such as when the user clicks a button, presses a key, or when an element loads.

When an event happens, we can execute something.

The most common events you might “listen out for” are `load`, `click`, `touchstart`, `mouseover`, `keydown`. You can check out all the DOM events in MDN's **Event Reference** guide.

How to Use Global Onevent Attributes in HTML

If you only want to add a one-liner script to a particular HTML element, you can use HTML's global onevent attributes defined by the HTML specification, such as onclick, onload, and onmouseover.

These attributes can be directly added to any HTML element that’s present on the page, however, their browser support widely varies. For instance, onclick is supported by all modern browsers up from IE9, while support for other onevent attributes such as ondrag is more patchy. You can check out browser support for global onevent attributes by typing “globaleventhandlers” into the search box on CanIUse.

The syntax of **onevent** attributes is simple and, as they are global attributes, you can use them on any element, for instance:

```
1   <button onclick="alert('Hi');">Click me</button>
```

Here, the onclick event listener listens to the click event on one specific button. When the event fires (the user clicks this button), the alert() callback function is executed.

If we want to add the same alert functionality to each button on the page, we should add the click event listener in a separate script rather than using the `onclick` attribute.

How to Create an Event Listener in JavaScript with addEventListener()

Using native JavaScript, we can listen to all the events defined in MDN's **Event Reference**, including touch events. As this doesn't require the use of a third-party library, it's the most performance-friendly solution to add interactive functionality to HTML elements.

We can create an event listener in JavaScript using the `addEventListener()` method that's **built into every modern browser**.

This is how our alert button example will look using plain JavaScript and the `addEventListener()` method:

```
/* Selecting DOM element */
const button = document.querySelector("button");

/* Callback function */
function alertButton() {
    alert('Hi native JavaScript');
}

/* Event listener */
button.addEventListener("click", alertButton, false);
```

Here it is in action:

In native JavaScript, we need to first select the DOM element that we want to add the event listener to.
The `querySelector()` method selects the first element that

matches a specified selector. So in our example, it selects the first `<button>` element on the page.

The custom `alertButton()` function is the callback function that will be called when the user clicks the button.

Finally, we add the event listener. We always have to attach the `addEventListener()` method to a pre-selected DOM element using the dot notation. In the parameters, first we define the event we want to listen to (`"click"`), then the name of the callback function (`alertButton`), finally the value of the `useCapture` parameter (we use the default `false` value, as we don't want to capture the event—**here's a simple explanation** about how to use `useCapture`).

How to Add Functionality to All Buttons

So, the code above adds the alert function to the first button on the page. But, how would we add the same functionality to *all* buttons? To do so, we need to use the **`querySelectorAll()`** method, loop through the elements, and add an event listener to each button:

```
/* Selecting DOM nodelist */
const buttons = document.querySelectorAll("button");

/* Callback function */
function alertButton() {
    alert('Hi native JavaScript');
}

/* Event listeners */
for (let button of buttons) {
      button.addEventListener("click", alertButton, false);
}
```

```
12
```

As `querySelectorAll()` returns a `NodeList` instead of a single element, we need to loop through the nodes to add a click event listener to each button. For instance, if we have three buttons on the page, the code above will create three click event listeners.

Note that you can only listen to one event with `addEventListener()`. So if you want the custom `alertButton()` function to fire on another event type such as `mouseover`, you'll need to create a second event listener rule:

```
/* Event listeners */
for (let button of buttons) {
      button.addEventListener("click", alertButton, false);
      button.addEventListener("mouseover", alertButton, false);
}
```

How to Combine Event Listeners with CSS and Conditionals

Probably the best thing about event listeners is that we can combine them with CSS and `if-else` conditional statements. In this way, we can target the different states of the same element with CSS and/or JavaScript.

For instance, here's a very simple example; a reveal-hide functionality. The HTML only consists of a button and a section. We will bind the section to the button using a JavaScript event listener. The button will be responsible for revealing and hiding the section below it:

```
<button class="reveal-button">Click me</button>
<section class="hidden-section">Lorem ipsum dolor sit amet...</secti
```

In the JavaScript, we first create two constants (`revealButton` and `hiddenSection`) for the two HTML elements using the `querySelector()` method.

Then, in the `revealSection()` callback function, we check if the hidden section has the `reveal` class or not using the **`classList`** property defined in the DOM API. If the hidden section *has* this class, we remove it using the DOM API's **`remove()`** method, and if it *doesn't*, we add it using the DOM API's **`add()`** method. Finally, we create an event listener for the click event.

```
/* Selecting DOM elements */
const revealButton = document.querySelector(".reveal-button");
const hiddenSection = document.querySelector(".hidden-section");

/* Callback function */
function revealSection() {
    if (hiddenSection.classList.contains("reveal")) {
        hiddenSection.classList.remove("reveal");
    } else {
        hiddenSection.classList.add("reveal");
    }
}

/* Event listener */
revealButton.addEventListener("click", revealSection, false);
```

Now, the JavaScript adds or removes the `.reveal` class depending on the current state of the hidden section. However, we still have to visually hide or reveal the element using CSS:

```
.hidden-section {
    display: none;
}
.hidden-section.reveal {
    display: block;
}
```

And, that's all! When the user first clicks the button, the hidden section is revealed, and when they click it the second time, it gets hidden again. You can test the functionality in the Codepen demo below: