**JavaScript Basics #7: Handling Events**

In computer programming, an event is a user input, such as mouse and keyboard actions, and the program we write is usually expected to respond with something. This process is called event handling.

Table of Contents

* [Event Handlers](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Event_Handlers)
  + [Register Event Handlers](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Register_Event_Handlers)
  + [Propagation](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Propagation)
  + [Default Actions](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Default_Actions)
* [Key Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Key_Events)
* [Pointer Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Pointer_Events)
  + [Mouse Clicks](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Mouse_Clicks)
  + [Mouse Motion](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Mouse_Motion)
  + [Touch Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Touch_Events)
* [Scroll Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Scroll_Events)
* [Focus Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Focus_Events)
* [Load Events](https://www.ericsdevblog.com/index.php/2021/11/javascript-basics-7-handling-events/#Load_Events)

**EVENT HANDLERS**

Let’s first take a look at a very simple example. We have an HTML document with a paragraph, and we want the page to return a message when it is clicked.

<p>Click this document to activate the handler.</p>

<script>

// Recall that the () => {} syntax is how we

define an arrow function in JavaScript

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

console.log("You knocked?");

});

</script>

This time, the output message will only appear in the console when you click on the document, instead of the moment the page is loaded.

**Register Event Handlers**

The addEventListener() method is how we can register an event handler for the document node. In fact, we can use the same method to register event handlers for any node in the HTML document. For example:

<!--This time we register a event handler for the button but not the paragraph-->

<button>Click me</button>

<p>No handler here.</p>

<script>

let button = document.querySelector("button");

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

console.log("Button clicked.");

});

</script>

Actually, there is an onclick attribute for the HTML nodes which will have the exact same effect. However, you can only register one handler for each node. By using the addEventListener() method, we are able to register multiple handlers for each node.

<button>Click me</button>

<script>

let button = document.querySelector("button");

// When you click the button, the console outputs "Button clicked."

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

console.log("Button clicked.");

});

// When you click the button, the console outputs "Button double clicked."

button.addEventListener("dblclick", () => {

console.log("Button double clicked.");

})

</script>

The removeEventListener() method, call with similar arguments can be used to remove an already registered event handler.

<button>Act-once button</button>

<script>

let button = document.querySelector("button");

function once() {

console.log("Done.");

button.removeEventListener("click", once);

}

button.addEventListener("click", once);

</script>

This button will only work once, after the removeEventListener("click", once) method is executed, the event handler registered for the button will be removed. The function that is passed to the removeEventListener has to be the same one that you passed to the addEventListener method.

**Propagation**

For most event types, the event handler registered for the node with children can receive events that happened in the children. For example, if a button inside a paragraph is clicked, the event handler registered for the paragraph will also be able to see that click event.

The event is said to propagate outward. For example, if both the button and the paragraph have an event handler, then the handler registered for the button will go first, then the paragraph, and it will keep propagating outward until it reaches the root of the document.

This feature can be quite useful sometimes, however, it is not always what we want. Luckily, we can stop the propagation using the stopPropagation() method.

<!--<button> is the child of <p>-->

<p>A paragraph with a <button>button</button>.</p>

<script>

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

let button = document.querySelector("button");

para.addEventListener("mousedown", () => {

console.log("Handler for paragraph.");

});

button.addEventListener("mousedown", event => {

console.log("Handler for button.");

// If the button is clicked with the right mouse button, there will be no propagation

if (event.button == 2) event.stopPropagation();

});

</script>

Sometimes we want to register event handlers for multiple elements on the page. To do this we can use the target attribute to cast a wide net for a type of event.

<button>A</button>

<button>B</button>

<button>C</button>

<script>

document.body.addEventListener("click", event => {

if (event.target.nodeName == "BUTTON") {

console.log("Clicked", event.target.textContent);

}

});

</script>

**Default Actions**

A lot of the events have a default action, for example, when you click on a link, you will be taken to the link’s target, if you press the down arrow, the browser will scroll the page down. You can prevent that default action from being activated by using the preventDefault() method. Let’s try something completely useless but very interesting.

<a href="https://developer.mozilla.org/">MDN</a>

<script>

let link = document.querySelector("a");

// When you click the link, instead of going to the URL that link specifies, the console will just output "Nope."

link.addEventListener("click", event => {

console.log("Nope.");

event.preventDefault();

});

</script>

Even though this is possible, do not do this unless you have a very good reason to. Or it will be very confusing for the users.

**KEY EVENTS**

Now we have discussed how event handlers work in general, it’s time to take a closer look at all the different types of events. The first one we are going to talk about is the key event.

When a key on your keyboard is pressed, it will trigger a keydown event, and when it is released, it triggers a keyup event.

<p>This page turns violet when you hold the V key.</p>

<script>

window.addEventListener("keydown", event => {

if (event.key == "v") {

document.body.style.background = "violet";

}

});

window.addEventListener("keyup", event => {

if (event.key == "v") {

document.body.style.background = "";

}

});

</script>

Looks very simple, however, you do need to be very careful about the keydown event. It is not a one-time thing, instead, it will keep being triggered over and over again, for as long as the key is being pressed, until it is released. You can experiment with the previous code, see what happens when you keep the key pressed.

There are also some special keys like CTRL, ALT, and SHIFT. These are called modifier keys, they modify the original value of other keys by forming a key combination. For instance, when you press a key while holding the SHIFT key, "s" will become "S", "1" will become "!" etc. We can register event handlers for key combinations like this:

<p>Press Control-Space to continue.</p>

<script>

window.addEventListener("keydown", event => {

if (event.key == " " && event.ctrlKey) {

console.log("Continuing!");

}

});

</script>

**POINTER EVENTS**

Pointer, as the name suggests, is used to point at things on the screen. There are primarily two ways that you can use to do that, either with a mouse or a touch screen. They produce different types of events.

**Mouse Clicks**

Mouse clicks work similarly to key events. When you press a mouse button, a mousedown event is triggered, and when you release that button, a mouseup event is triggered. And after the mouseup event, a complete click is finished, so a click event will be fired.

<button>Click me!</button>

<script>

let button = document.querySelector("button");

button.addEventListener("mousedown", event => {

console.log("mouse down");

});

button.addEventListener("mouseup", event => {

console.log("mouse up");

});

button.addEventListener("click", event => {

console.log("button clicked");

});

</script>

When two clicks happen very close together, a dblclick (double click) event will be triggered after the second click.

<button>Double click me!</button>

<script>

let button = document.querySelector("button");

button.addEventListener("dblclick", (event) => {

console.log("double clicked");

});

</script>

**Mouse Motion**

When a mouse pointer moves, a mousemove event is triggered.

<p>Move the cursor onto this paragraph to turn it red.</p>

<script>

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

para.addEventListener("mousemove", (event) => {

para.style.color = "red";

});

</script>

This can be very useful when you are trying to implement some sort of drag and drop functionality. But to do that, we need to first track the location of the cursor. To get that information, we can either use the event’s clientX and clientY properties, which contains the event’s coordinates (in pixels) relative to the top-left corner of the window, or pageX and pageY, which are relative to the top-left corner of the whole document.

For example, the following script will output the coordinates of the click events that happened on the page.

<p>click anywhere</p>

<script>

window.addEventListener("click", event => {

console.log("X: " + event.clientX);

console.log("Y: " + event.clientY);

});

</script>

Here is a more complicated example, this program will display a bar, and you can drag it to change its width.

<p>Drag the bar to change its width:</p>

<div style="background: orange; width: 60px; height: 20px">

</div>

<script>

let lastX; // Tracks the last observed mouse X position

let bar = document.querySelector("div");

bar.addEventListener("mousedown", event => {

if (event.button == 0) { // if the left button is being held

lastX = event.clientX;

// If the cursor moves while the left button is being held

window.addEventListener("mousemove", moved);

event.preventDefault(); // Prevent selection

}

});

function moved(event) {

// If no button is being held, remove the "mousemove" event handler

if (event.buttons == 0) { // Notice this is "buttons" not "button"

window.removeEventListener("mousemove", moved);

} else {

let dist = event.clientX - lastX;

let newWidth = Math.max(10, bar.offsetWidth + dist);

bar.style.width = newWidth + "px";

lastX = event.clientX;

}

}

</script>

Notice that we used two different ways to access which button is pushed (The button property and the buttons property), and they clearly work differently. Their main difference is that the button property can only tell you which button (singular) is clicked, while the buttons property can tell you if a combination of buttons is pushed.

**The button property**

* 0: Main button pressed, usually the left button or the un-initialized state
* 1: Auxiliary button pressed, usually the wheel button or the middle button (if present)
* 2: Secondary button pressed, usually the right button
* 3: Fourth button, typically the *Browser Back* button
* 4: Fifth button, typically the *Browser Forward* button

**The buttons property**

* 0: No button or un-initialized
* 1: Primary button (usually the left button)
* 2: Secondary button (usually the right button)
* 4: Auxiliary button (usually the mouse wheel button or middle button)
* 8: 4th button (typically the “Browser Back” button)
* 16 : 5th button (typically the “Browser Forward” button)

When more than one button is pressed simultaneously, the values are combined. For example, when the primary and secondary buttons are pressed at the same time, the value will be 3.

**Touch Events**

In most cases, the mouse events will also work when the user is using a touch screen. For example, when you are tapping a button on your screen, it will trigger a click event, it will be the same as clicking it with a mouse pointer.

However, this won’t work in some cases, such as the resizing bar example we talked about before. Because the touch screen doesn’t have multiple buttons, and it can’t track your finger’s position when you are not touching the screen. So to solve this problem, we have a few specific event types triggered only by touch interaction.

When your finger touches the screen, it triggers a touchstart event, when it moves while touching, it triggers a touchmove event, and finally, when you lift your finger, it triggers a touchend event.

**SCROLL EVENTS**

A scroll event is triggered when you place the cursor on an element and scroll the middle button of your mouse. This can be very useful when you are trying to make your webpage more responsive. For example, when you go to the product showcasing page on Apple’s website, notice that the elements on the page will move as you scroll down.

Here is an example of a progress bar, it starts at 0% and will go to 100% as you scroll down.

<style>

#progress {

border-bottom: 20px solid orange;

width: 0;

position: fixed;

top: 0; left: 0;

}

</style>

<div id="progress"></div>

<script>

// Create some content

document.body.appendChild(document.createTextNode(

"supercalifragilisticexpialidocious ".repeat(1000)));

let bar = document.querySelector("#progress");

window.addEventListener("scroll", () => {

let max = document.body.scrollHeight - innerHeight;

bar.style.width = `${(pageYOffset / max) \* 100}%`;

});

</script>

**FOCUS EVENTS**

When an element gains focus, a focus event will be triggered, and when the element loses focus, a blur event will be triggered. Unlike the other event types we’ve discussed, these two do not propagate.

This is most commonly used on HTML field elements. When you click on a text field and start typing some texts, that field is said to be in focus, and when you move on from that field and click on other elements, that field element loses focus.

This is an example that displays help texts for the text field that is currently in focus.

<p>Name: <input type="text" data-help="Your full name"></p>

<p>Age: <input type="text" data-help="Your age in years"></p>

<p id="help"></p>

<script>

let help = document.querySelector("#help");

let fields = document.querySelectorAll("input");

for (let field of Array.from(fields)) {

field.addEventListener("focus", event => {

let text = event.target.getAttribute("data-help");

help.textContent = text;

});

field.addEventListener("blur", event => {

help.textContent = "";

});

}

</script>

**LOAD EVENTS**

The load event is triggered when the entire page finishes loading. This is different from directly putting the code inside the <script> tag directly without event handlers. The code inside the <script> tag is run immediately when the tag is encountered. This might be too soon in some cases.

There is also a similar event type called beforeunload. It is triggered when you close a page, the primary use of this event is to prevent the user from accidentally closing their unsaved work.