Introduction to events

Events are things that happen in the system you are programming, which the system tells you about so your code can react to them. For example, if the user clicks a button on a webpage, you might want to react to that action by displaying an information box. In this article, we discuss some important concepts surrounding events, and look at the fundamentals of how they work in browsers.

Prerequisites:	An understanding of HTML and the fundamentals of CSS, familiarity with JavaScript basics as covered in previous lessons.
Learning outcomes:	• What events are — a signal fired by the browser when something significant happens, which the developer can run some code in response to. • Setting up event handlers using `addEventListener()` (and `removeEventListener()`) and event handler properties. • Inline event handler attributes, and why you shouldn't use them. • Event objects.

What is an event?

Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. 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, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur.

For example:

- The user selects, clicks, or hovers the cursor over a certain element.
- The user presses 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.

You can gather from this (and from glancing at the MDN event reference) that there are **a lot** of events that can be fired.

To react to an event, you attach an **event handler** to it. This is a block of code (usually a JavaScript 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 for our purposes, although strictly speaking, they work together. The listener listens out for the event happening, and the handler is the code that runs in response to it happening.

> **Note:** Web events are not part of the core JavaScript language — they are defined as part of the APIs built into the browser.

An example: handling a click event

In the following example, we have a single `<button>` in the page:

HTML Play

```html
<button>Change color</button>
```

Then we have some JavaScript. We'll look at this in more detail in the next section, but for now we can just say: it adds an event handler to the button's `"click"` event, and the handler reacts to the event by setting the page background to a random color:

JS Play

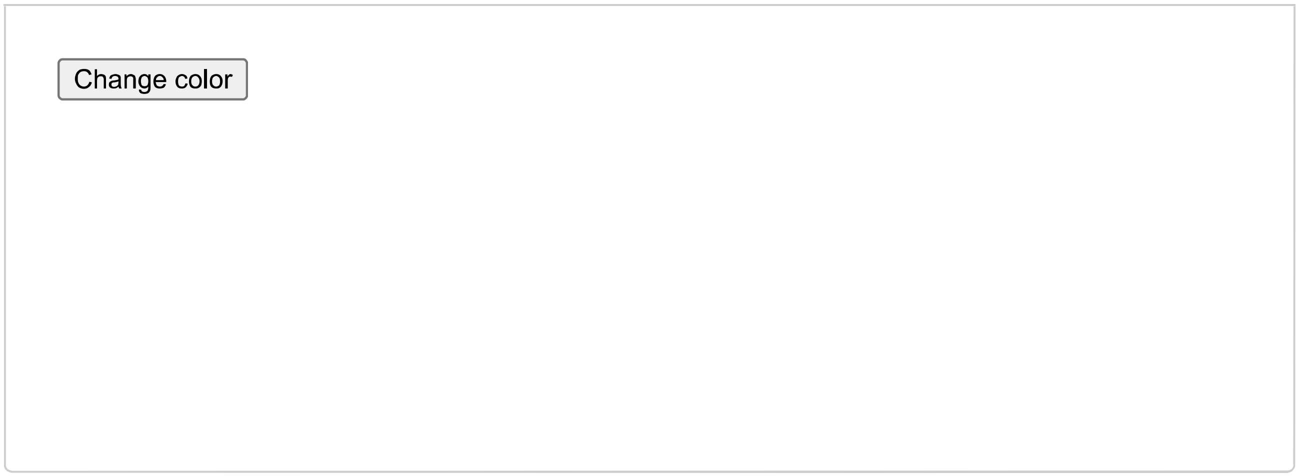
```js
const btn = document.querySelector("button");

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;
});
```

The example output is as follows. Try clicking the button:

Play



Using addEventListener()

As we saw in the last example, objects that can fire events have an `addEventListener()` method, and this is the recommended mechanism for adding event handlers.

Let's take a closer look at the code from the last example:

JS

```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;
});
```

The HTML `<button>` element will fire an event when the user clicks the button. So it defines an `addEventListener()` function, which we are calling here. We're passing in two parameters:

- the string `"click"`, to indicate that we want to listen to the click event. Buttons can fire lots of other events, such as `"mouseover"` when the user moves their mouse over the button, or `"keydown"` when the user presses a key and the button is focused.
- a function to call when the event happens. In our case, the function generates a random RGB color and sets the `background-color` of the page `<body>` to that color.

It is fine to make the handler function a separate named function, like this:

JS

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

```
function random(number) {
  return Math.floor(Math.random() * (number + 1));
}

function changeBackground() {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}

btn.addEventListener("click", changeBackground);
```

Listening for other events

There are many different events that can be fired by a button element. Let's experiment.

First, make a local copy of random-color-addeventlistener.html , and open it in your browser. It's just a copy of the simple random color example we've played with already. Now try changing `click` to the following different values in turn, and observing the results in the example:

- `focus` and `blur` — The color changes when the button is focused and unfocused; try pressing the tab to focus on the button and press the tab again to focus away from the button. These are often used to display information about filling in form fields when they are focused, or to display an error message if a form field is filled with an incorrect value.
- `dblclick` — The color changes only when the button is double-clicked.
- `mouseover` and `mouseout` — The color changes when the mouse pointer hovers over the button, or when the pointer moves off the button, respectively.

Some events, such as `click`, are available on nearly any element. Others are more specific and only useful in certain situations: for example, the `play` event is only available on some elements, such as `<video>`.

Removing listeners

If you've added an event handler using `addEventListener()`, you can remove it again using the `removeEventListener()` method. For example, this would remove the `changeBackground()` event handler:

JS

```js
btn.removeEventListener("click", changeBackground);
```

Event handlers can also be removed by passing an `AbortSignal` to `addEventListener()` and then later calling `abort()` on the controller owning the `AbortSignal`. For example, to add an event handler that we can remove with an `AbortSignal`:

JS

```js
const controller = new AbortController();

btn.addEventListener("click",
  () => {
    const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
    document.body.style.backgroundColor = rndCol;
  },
  { signal: controller.signal } // pass an AbortSignal to this handler
);
```

Then the event handler created by the code above can be removed like this:

JS

```js
controller.abort(); // removes any/all event handlers associated with this controller
```

For simple, small programs, cleaning up old, unused event handlers isn't necessary, but for larger, more complex programs, it can improve efficiency. Also, the ability to remove event handlers allows you to have the same button performing different actions in different circumstances: all you have to do is add or remove handlers.

Adding multiple listeners for a single event

By making more than one call to `addEventListener()`, providing different handlers, you can have multiple handlers for a single event:

JS

```js
myElement.addEventListener("click", functionA);
myElement.addEventListener("click", functionB);
```

Both functions would now run when the element is clicked.

Other event listener mechanisms

We recommend that you use `addEventListener()` to register event handlers. It's the most powerful method and scales best with more complex programs. However, there are two other ways of registering event handlers that you might see: *event handler properties* and *inline event handlers*.

Event handler properties

Objects (such as buttons) that can fire events also usually have properties whose name is `on` followed by the name of the event. For example, elements have a property `onclick`. This is called an *event handler property*. To listen for the event, you can assign the handler function to the property.

For example, we could rewrite the random-color example like this:

JS

```js
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;
};
```

You can also set the handler property to a named function:

JS

```
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;
```

With event handler properties, you can't add more than one handler for a single event. For example, you can call `addEventListener('click', handler)` on an element multiple times, with different functions specified in the second argument:

JS

```
element.addEventListener("click", function1);
element.addEventListener("click", function2);
```

This is impossible with event handler properties because any subsequent attempts to set the property will overwrite earlier ones:

JS

```
element.onclick = function1;
element.onclick = function2;
```

Inline event handlers — don't use these

You might also see a pattern like this in your code:

HTML

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

JS

```js
function bgChange() {
  const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
}
```

The earliest method of registering event handlers found on the Web involved *event handler HTML attributes* (or *inline event handlers*) like the one shown above — the attribute value is literally the JavaScript code you want to run when the event occurs. The above example invokes a function defined inside a `<script>` element on the same page, but you could also insert JavaScript directly inside the attribute, for example:

HTML

```html
<button onclick="alert('Hello, this is my old-fashioned event handler!');">
  Press me
</button>
```

You can find HTML attribute equivalents for many of the event handler properties; however, you shouldn't use these — they are considered bad practice. It might seem easy to use an event handler attribute if you are doing something really quick, but they quickly become unmanageable and inefficient.

For a start, it is not a good idea to mix up your HTML and your JavaScript, as it becomes hard to read. Keeping your JavaScript separate is a good practice, and if it is in a separate file you can apply it to multiple HTML documents.

Even in a single file, inline event handlers are not a good idea. One button is OK, but what if you had 100 buttons? You'd have to add 100 attributes to the file; it would quickly turn into a maintenance nightmare. With JavaScript, you could easily add an event handler function to all the buttons on the page no matter how many there were, using something like this:

JS

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", bgChange);
}
```

Finally, many common server configurations will disallow inline JavaScript, as a security measure.

You should never use the HTML event handler attributes — those are outdated, and using them is bad practice.

Event objects

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. For example, let's rewrite our random color example again slightly:

JS

```
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);
```

> **Note:** You can find the full source code for this example on GitHub (also see it running live).

Here you can see we are including an event object, **e**, in the function, and in the function setting a background color style on `e.target` — which is the button itself. The `target` property of the event object is always a reference to the element the event occurred upon. So, in this example, we are setting a random background color on the button, not the page.

> **Note:** You can use any name you like for the event object — you just need to choose a name that you can then use to reference it inside the event handler function. `e` / `evt` / `event` is most commonly used by developers because they are short and easy to remember. It's always good to be consistent — with yourself, and with others if possible.

Extra properties of event objects

Most event objects have a standard set of properties and methods available on the event object; see the `Event` object reference for a full list.

Some event objects add extra properties that are relevant to that particular type of event. For example, the `keydown` event fires when the user presses a key. Its event object is a `KeyboardEvent` , which is a specialized `Event` object with a `key` property that tells you which key was pressed:

HTML Play

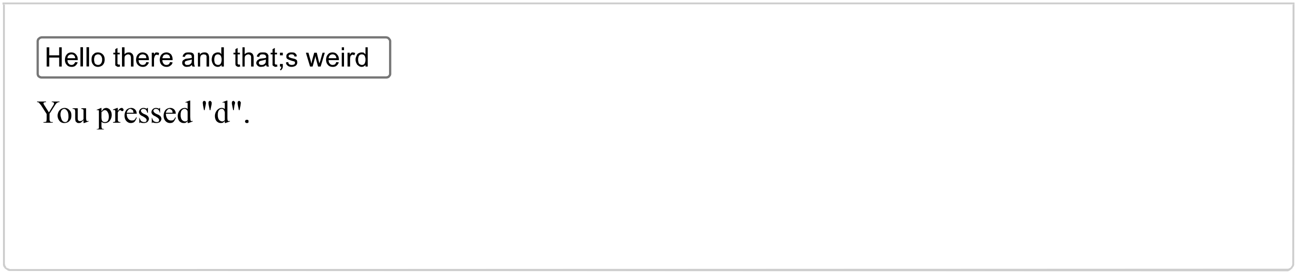
```html
<input id="textBox" type="text" />
<div id="output"></div>
```

JS Play

```js
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");
textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
});
```

Try typing into the text box and see the output:

Play



Hello there and that;s weird

You pressed "d".

Preventing default behavior

Sometimes, you'll come across a situation 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 example, a custom registration form. When you fill in the details and click the submit button, the natural behavior is for the data to be submitted to a specified page on the server for processing, and the browser to be redirected to a "success message" page of some kind (or the same page, if another is not specified).

The trouble comes when the user has not submitted the data correctly — as a developer, you want to prevent the submission to the server and give an error message saying what's wrong and what needs to be done to put things right. Some browsers support automatic form data validation features, but since many don't, you are advised to not rely on those and implement your own validation checks. Let's look at an example.

First, a simple HTML form that requires you to enter your first and last name:

HTML Play

```html
<form>
  <div>
    <label for="fname">First name: </label>
    <input id="fname" type="text" />
  </div>
  <div>
    <label for="lname">Last name: </label>
    <input id="lname" type="text" />
  </div>
  <div>
    <input id="submit" type="submit" />
  </div>
</form>
<p></p>
```

Now some JavaScript — here we implement a very simple check inside a handler for the `submit` event (the submit event is fired on a form when it is submitted) that tests whether the text fields are empty. If they are, we call the `preventDefault()` function on the event object — which stops the form submission — and then display an error message in the paragraph below our form to tell the user what's wrong:

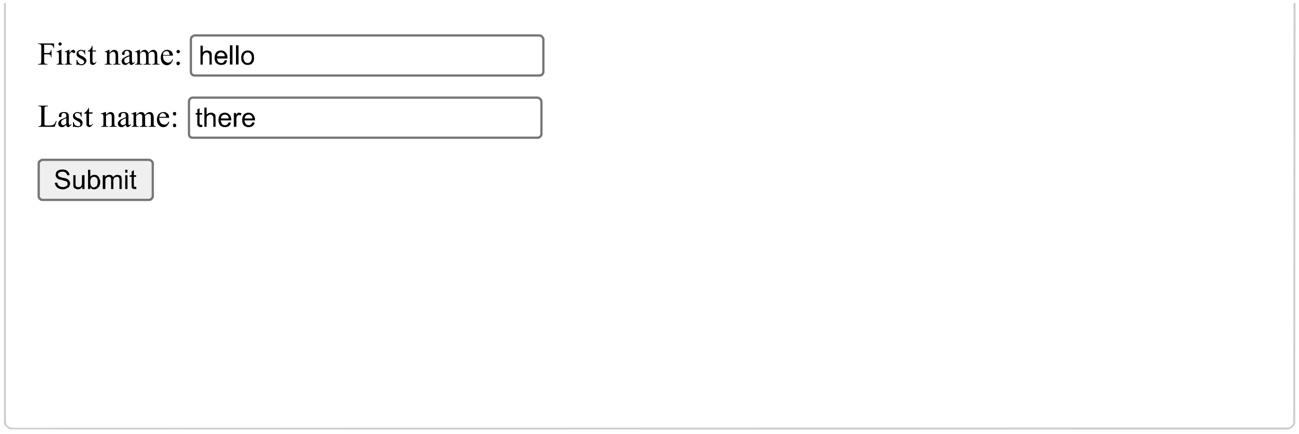
JS Play

```js
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!";
  }
});
```

Obviously, this is pretty weak form validation — it wouldn't stop the user from validating the form with spaces or numbers entered into the fields, for example — but it is OK for example purposes. The output is as follows:

Play


First name: hello

Last name: there

Submit

Note: For the full source code, see preventdefault-validation.html (also see it running live here).

It's not just web pages

Events are not unique to JavaScript — most programming languages have some kind of event model, and the way the model works often differs from JavaScript's way. In fact, the event model in JavaScript for web pages differs from the event model for JavaScript as it is used in other environments.

For example, Node.js is a very popular JavaScript runtime that enables developers to use JavaScript to build network and server-side applications. The Node.js event model relies on listeners to listen for events and emitters to emit events periodically — it doesn't sound that different, but the code is quite different, making use of functions like `on()` to register an event listener, and `once()` to register an event listener that unregisters after it has run once. The HTTP connect event docs provide a good example.

You can also use JavaScript to build cross-browser add-ons — browser functionality enhancements — using a technology called WebExtensions. The event model is similar to the web events model, but a bit different — event listeners' properties are written in camel case (such as `onMessage` rather than `onmessage`), and need to be combined with the `addListener` function. See the `runtime.onMessage` page for an example.

You don't need to understand anything about other such environments at this stage in your learning; we just wanted to make it clear that events can differ in different programming environments.

Summary

In this chapter we've learned what events are, how to listen for events, and how to respond to them.

You've seen by now that elements in a web page can be nested inside other elements. For example, in the Preventing default behavior example, we have some text boxes, placed inside `<div>` elements, which in turn are placed inside a `<form>` element. What happens when a click event listener is attached to the `<form>` element, and the user clicks inside one of the text boxes? The associated event handler function is still fired via a process called *event bubbling*, which is covered in the next lesson.