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 how they work in browsers. This won't be an exhaustive study; just what you need to know at this stage.

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. There are more than 280 different events in different web technologies, libraries, or APIs.

Here are some commonly used events:

Mouse events	`click, mouseover, mouseout, mousedown, mouseup`
Keyboard events	`keydown, keyup, keypress`
Form events	`submit, input, change, focus, blur`
Window events	`load, resize, scroll, unload`
Touch events	`touchstart, touchmove, touchend`
Media events	`play ,pause, ended`
Drag and drop events	`dragstart, dragenter, dragover, drop, dragend`

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 is run in response to it happening.

An example: handling a click event

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

```
<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:

```
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;
});
```

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:

```
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:

```
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. Let's take our previous example, and 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. 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: for example, the `play` event is 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:

```
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`:

```
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:

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

For simple, small programs, cleaning up old, unused event handlers aren'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()`, and providing different handlers, you can have multiple handlers for a single event:

```
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:

```
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:

```
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:

```
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:

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

Inline event handlers — don't use these

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

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

```
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:

```
<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:

```
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:

```
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);
```

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.

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:

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

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

In this example when you type in a box and in text below your last pressed key will be visible.

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. 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 many don't.

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

```
<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>
```

Here we implement 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:

```
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.

Event bubbling

Event bubbling is a mechanism in JavaScript where when an event is triggered on a nested element, the same event is also triggered on its parent elements all the way up to the root element. It's like a bubble that rises up from the innermost element to the outermost ancestor. This allows you to handle events at higher levels in the DOM hierarchy without attaching individual event listeners to each nested element.

Let's go through an example of event bubbling:

```
<div id="outer">
  <div id="middle">
    <div id="inner">Click me!</div>
  </div>
</div>
```

```
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener("click", function (event) {
  console.log("Inner element clicked!");
});

middleElement.addEventListener("click", function (event) {
  console.log("Middle element clicked!");
});

outerElement.addEventListener("click", function (event) {
  console.log("Outer element clicked!");
});
```

In this example, when you click on the "Click me!" text, you will see the following output in the console:

```
Inner element clicked!
Middle element clicked!
Outer element clicked!
```

Fixing the problem with stopPropagation()

Now, let's talk about `stopPropagation()`. It is a method available on the event object, and when called within an event handler, it **prevents the event from bubbling up to its parent elements**. This means that if `stopPropagation()` is used within the innermost element's event handler, the event won't reach the middle and outer elements.

Here's the modified code using `stopPropagation()`:

```
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener("click", function (event) {
  console.log("Inner element clicked!");
  event.stopPropagation(); // Stops the event from bubbling up further
});

middleElement.addEventListener("click", function (event) {
  console.log("Middle element clicked!");
});

outerElement.addEventListener("click", function (event) {
  console.log("Outer element clicked!");
});
```

Now, when you click on the "Click me!" text, you will see the following output in the console:

```
Inner element clicked!
```

The event stops at the innermost element and doesn't bubble up to the middle and outer elements.

Event capture

As for Event Capture, it is an alternative event handling phase to event bubbling. In Event Capture, the event is first captured at the root element and then propagated down to the target element. It is the **reverse of event bubbling**.

Here's an example using Event Capture:

```
const innerElement = document.getElementById("inner");
const middleElement = document.getElementById("middle");
const outerElement = document.getElementById("outer");

innerElement.addEventListener(
  "click",
  function (event) {
    console.log("Inner element clicked!");
  },
  true // Adding 'true' as the third parameter enables Event Capture
);

middleElement.addEventListener(
  "click",
  function (event) {
    console.log("Middle element clicked!");
  },
  true
);

outerElement.addEventListener(
  "click",
  function (event) {
    console.log("Outer element clicked!");
  },
  true
);
```

Now, when you click on the "Click me!" text, you will see the following output in the console:

```
Outer element clicked!
Middle element clicked!
Inner element clicked!
```

The event starts capturing from the outermost element and propagates down to the innermost element, giving you the reverse order compared to event bubbling.

Event Capture is less commonly used than event bubbling, but it provides an alternative way to handle events in specific scenarios where you need to intercept events as they flow down the DOM hierarchy.

Event delegation

Event delegation is a design pattern in JavaScript where **instead of attaching event listeners to individual elements, you attach a single event listener to a parent element** that encompasses all the child elements you are interested in. Then, you use event bubbling to handle the events as they propagate up to the parent element. This approach is particularly useful when you have a large number of elements with similar behavior, as it helps improve performance and simplifies event handling code.

Here's an example to illustrate event delegation:

```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

```
const list = document.getElementById("myList");

list.addEventListener("click", function (event) {
  if (event.target.tagName === "LI") {
    // Check if the clicked element is an <li> element
    console.log("Clicked on:", event.target.textContent);
  }
});
```

In this example, we have a list `(<ul>)` with multiple list items `(<li>).` Instead of attaching individual event listeners to each list item, we attach a single event listener to the parent `<ul>` element. When you click on any list item, the event bubbles up to the `<ul>` element, and the event listener checks if the clicked target (`event.target`) is an `<li>` element. If it is, it logs the text content of the clicked list item to the console.

The event delegation technique here allows us to handle clicks on any list item without adding separate event listeners to each of them. This is particularly beneficial when dealing with dynamic content where new list items can be added or removed, and you don't need to worry about attaching new event listeners to the new elements.

Event delegation is a powerful technique that promotes cleaner and more efficient code, especially when working with large sets of similar elements. However, it's essential to ensure that the parent element chosen for delegation is stable and present in the DOM at the time of attaching the event listener.

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.

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 camel-cased (such as `onMessage` rather than `onmessage`), and need to be combined with the `addListener` function.