JavaScript Essentials

Events and Listeners

Table of Contents

Lesson Objectives

- Understand the fundamental theory of events, how they work in browsers, and how events may differ in different programming environments
- Able to handle Event using inline event handler or addEventListener() method
- Able to capture user input using Event handler
- Combine with Selector API to create interactive web app

Section 1

Events

Events

- Events are actions or occurrences that happen in the system you are programming, which the system tells you about so you can respond to them in some way if desired.
- For example, if the user clicks a button on a webpage, you might want to respond 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.

Events

- As mentioned above, **events** are actions or occurrences that happen in the system you are programming
- The system produces (or "fires") a signal of some kind when an event occurs, and also provides a mechanism by which some kind of action can be automatically taken (that is, some code running) when the event occurs.
- For example in an airport when the runway is clear for a plane to take off, a signal is communicated to the pilot, and as a result, they commence piloting the plane.

Events

- In the case of the Web, 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, set of elements, the HTML document loaded in the current tab, or the entire browser window. There are a lot of different types of events that can occur, for example:
  - The user clicking the mouse over a certain element or hovering the cursor over a certain element.
  - The user pressing a key on the keyboard.
  - The user resizing or closing the browser window.
  - A web page finishing loading.
  - A form being submitted.
  - A video being played, or paused, or finishing play.
  - An error occurring.

Events

- Each available event has an **event handler**, which is a block of code (usually a JavaScript function that you as a programmer create) that will be run when the event fires.
- When such a block of code is defined to be run in response to an event firing, we say we are **registering an event handler**.
- Note that 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.

Events - Example

The example output is as follows:

Events – Summary

- **Events** are actions or occurrences that happen in the system you are programming
- Events are fired inside the browser window, and tend to be attached to a specific item that resides in it
- Each available event has an **event handler**
- Web events are not part of the core JavaScript language — they are defined as part of the APIs built into the browser

Section 2

Using Web Events

Using Web Events - Event handler properties

- There are a number of ways in which you can add event listener code to web pages so that it will be run when the associated event fires. In this section, we review the various mechanisms and discuss which ones you should use.
- These are the properties that exist to contain event handler code that we have seen most frequently during the course. Returning to the above example:

```
const btn = document.querySelector('button');

btn.onclick = function() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

Using Web Events - Event handler properties

- The onclick property is the event handler property being used in this situation.
- It is essentially a property like any other available on the button (e.g. btn.textContent, or btn.style), but it is a special type — when you set it to be equal to some code, that code is run when the event fires on the button.
- You could also set the handler property to be equal to a named function name (like we saw in Build your own function). The following would work just the same:

```
1  const btn = document.querySelector('button');
2
3  function bgChange() {
4    const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
5    document.body.style.backgroundColor = rndCol;
6  }
7
8  btn.onclick = bgChange;
```

Practice Event handler properties

Using Web Events - Inline Event handler

- 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;
}
```

Using Web Events - Inline Event handler

- 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:

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

Using Web Events - Inline Event handler

- 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 just doing something really quick, but they very 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 parse — keeping your JavaScript all in one place is better; if it is in a separate file you can apply it to multiple HTML documents.

Using Web Events - Inline Event handler

- 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 very 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:

```
1  const buttons = document.querySelectorAll('button');
2
3  for (let i = 0; i < buttons.length; i++) {
4    buttons[i].onclick = bgChange;
5  }
```

Using Web Events – addEventListener()

- The newest type of event mechanism is defined in the Document Object Model (DOM) Level 2 Events Specification, which provides browsers with a new function — addEventListener(). This functions in a similar way to the event handler properties, but the syntax is obviously different. We could rewrite our random color example to look like this:

```
const btn = document.querySelector('button');

function bgChange() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', bgChange);
```

Using Web Events – addEventListener()

- Inside the addEventListener() function, we specify two parameters — the name of the event we want to register this handler for, and the code that comprises the handler function we want to run in response to it. Note that it is perfectly appropriate to put all the code inside the addEventListener() function, in an anonymous function, like this:

```
1  btn.addEventListener('click', function() {
2    var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
3    document.body.style.backgroundColor = rndCol;
4  });
```

Using Web Events – addEventListener()

- This mechanism has some advantages over the older mechanisms discussed earlier. For a start, there is a counterpart function, removeEventListener(), which removes a previously added listener. For example, this would remove the listener set in the first code block in this section:

```
1 | btn.removeEventListener('click', bgChange);
```

Using Web Events – addEventListener()

- This isn't significant for simple, small programs, but for larger, more complex programs it can improve efficiency to clean up old unused event handlers. Plus, for example, this allows you to have the same button performing different actions in different circumstances — all you have to do is add or remove event handlers as appropriate.
- Second, you can also register multiple handlers for the same listener. The following two handlers wouldn't both be applied:

```
1 | myElement.onclick = functionA;
2 | myElement.onclick = functionB;
```

Using Web Events – addEventListener()

- The second line overwrites the value of onclick set by the first line. This would work, however:

```
1 myElement.addEventListener('click', functionA);
2 myElement.addEventListener('click', functionB);
```

- Both functions would now run when the element is clicked.
- In addition, there are a number of other powerful features and options available with this event mechanism. These are a little out of scope for this article, but if you want to read up on them, have a look at the addEventListener() and removeEventListener() reference pages.

Using Web Events – What mechanism to use?

- Of the three mechanisms, you definitely shouldn't use the HTML event handler attributes — these are outdated, and bad practice, as mentioned above.
- The other two are relatively interchangeable, at least for simple uses:
  - Event handler properties have less power and options, but better cross-browser compatibility (being supported as far back as Internet Explorer 8). You should probably start with these as you are learning.
  - DOM Level 2 Events (addEventListener(), etc.) are more powerful, but can also become more complex and are less well supported (supported as far back as Internet Explorer 9). You should also experiment with these, and aim to use them where possible.

Using Web Events – What mechanism to use?

- The main advantages of the third mechanism are that you can remove event handler code if needed, using removeEventListener(), and you can add multiple listeners of the same type to elements if required. For example, you can call addEventListener('click', function() { ... }) on an element multiple times, with different functions specified in the second argument. This is impossible with event handler properties because any subsequent attempts to set a property will overwrite earlier ones, e.g.:

```
1 element.onclick = function1;
2 element.onclick = function2;
3 etc.
```

Using Web Events - Overview

- There are three ways you can use to handle Event in Web: Inline handler, Handler properties and addEventListener()
- Definitely shouldn't use the inline event handler attributes — these are outdated, and bad practice
- The other two are relatively interchangeable
- For now, use addEventListener()

Section 3

Events object

Events object

- Sometimes inside an event handler function, you might see a parameter specified with a name such as event, evt, or simply 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:

```
function bgChange(e) {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  e.target.style.backgroundColor = rndCol;
  console.log(e);
}

btn.addEventListener('click', bgChange);
```

Events object

- Here you can see that 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 that the event has just occurred upon. So in this example, we are setting a random background color on the button, not the page.
- 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 are 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.

Events object

- e.target is incredibly useful when you want to set the same event handler on multiple elements and do something to all of them when an event occurs on them.
- You might, for example, have a set of 16 tiles that disappear when they are clicked on. It is useful to always be able to just set the thing to disappear as e.target, rather than having to select it in some more difficult way.
- In the following example (see useful-eventtarget.html for the full source code; also see it running live here), we create 16 elements using JavaScript.
- We then select all of them using document.querySelectorAll(), then loop through each one, adding an onclick handler to each that makes it so that a random color is applied to each one when clicked

Events object – Practice time

The output is as follows (try clicking around on it — have fun):

Events object – Prevent 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 press 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 a simple example.

Events object – Prevent default behavior

- 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>
```

Events object – Event bubbling and capture

- The final subject to cover here is something that you won't come across often, but it can be a real pain if you don't understand it. Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element. Let's look at an example to make this easier — open up the show-video-box.html example in a new tab (and the source code in another tab.) It is also available live below:
- But there's a problem — currently, when you click the video it starts to play, but it causes the to also be hidden at the same time. This is because the video is inside the — it is part of it — so clicking on the video actually runs *both* the above event handlers.

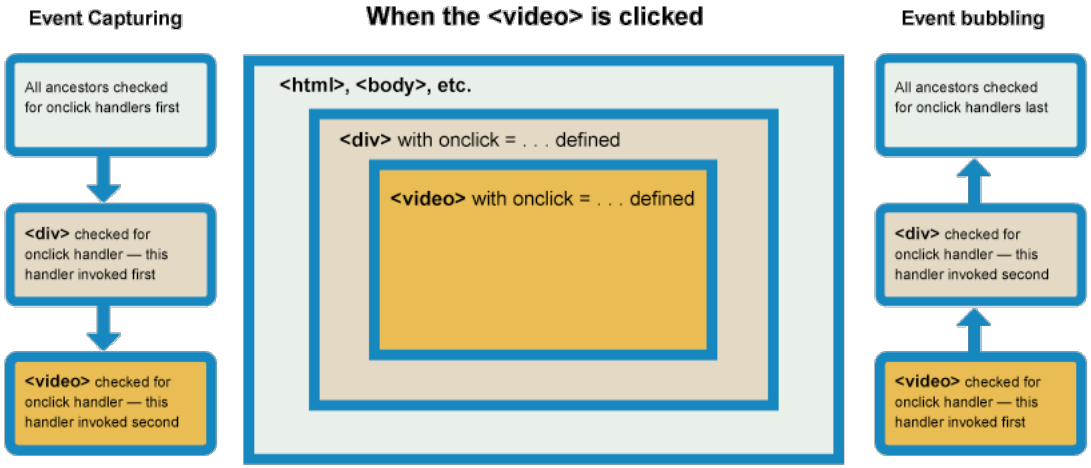
Events object – Event bubbling and capture

- When an event is fired on an element that has parent elements (in this case, the <video> has the as a parent), modern browsers run two different phases — the **capturing** phase and the **bubbling** phase.
- In the **capturing** phase:
  - ✓ The browser checks to see if the element's outer-most ancestor (<html>) has an onclick event handler registered on it for the capturing phase, and runs it if so.
  - ✓ Then it moves on to the next element inside <html> and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

Events object – Event bubbling and capture

- In the **bubbling** phase, the exact opposite occurs:
  - The browser checks to see if the element that was actually clicked on has an onclick event handler registered on it for the bubbling phase, and runs it if so.
  - Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the <html> element.

Events object – Event bubbling and capture



Events object – Event bubbling and capture

- In modern browsers, by default, all event handlers are registered for the bubbling phase. So in our current example, when you click the video, the click event bubbles from the <video> element outwards to the <html> element. Along the way:
- It finds the video.onclick... handler and runs it, so the video first starts playing.
- It then finds the videoBox.onclick... handler and runs it, so the video is hidden as well.
- In cases where both types of event handlers are present, bubbling and capturing, the capturing phase will run first, followed by the bubbling phase.

Events object – Event bubbling and capture

- This is annoying behavior, but there is a way to fix it! The standard Event object has a function available on it called stopPropagation() which, when invoked on a handler's event object, makes it so that first handler is run but the event doesn't bubble any further up the chain, so no more handlers will be run.
- We can, therefore, fix our current problem by changing the second handler function in the previous code block to this

```
1 video.onclick = function(e) {
2   e.stopPropagation();
3   video.play();
4 };
```

Events object – Event bubbling and capture

- Why bother with both capturing and bubbling? Well, in the bad old days when browsers were much less cross-compatible than they are now, Netscape only used event capturing, and Internet Explorer used only event bubbling. When the W3C decided to try to standardize the behavior and reach a consensus, they ended up with this system that included both, which is the one modern browsers implemented.
- As mentioned above, by default all event handlers are registered in the bubbling phase, and this makes more sense most of the time. If you really want to register an event in the capturing phase instead, you can do so by registering your handler using addEventListener(), and setting the optional third property to **true**.

Events object – Event bubbling and capture

- Bubbling also allows us to take advantage of **event delegation** — this concept relies on the fact that if you want some code to run when you click on any one of a large number of child elements, you can set the event listener on their parent and have events that happen on them bubble up to their parent rather than having to set the event listener on every child individually. Remember earlier that we said bubbling involves checking the element the event is fired on for an event handler first, then moving up to the element's parent, etc.?
- A good example is a series of list items — if you want each one of them to pop up a message when clicked, you can set the click event listener on the parent <ul>, and events will bubble from the list items to the <ul>.

Practice Events object

Events Object - Overview

- **Event object** it is automatically passed to event handlers to provide extra features and information
- The target property of the event object is always a reference to the element that the event has just occurred upon
- e.target is incredibly useful when you want to set the same event handler on multiple elements and do something to all of them when an event occurs on them
- Use event.preventDefault() method to prevent an event from doing what it does by default
- Modern Browser supports Event capturing and Event bubbling mode (default)
- Take advantage of **Event Delegation** to write less code but do more task

Thank you

Q&A