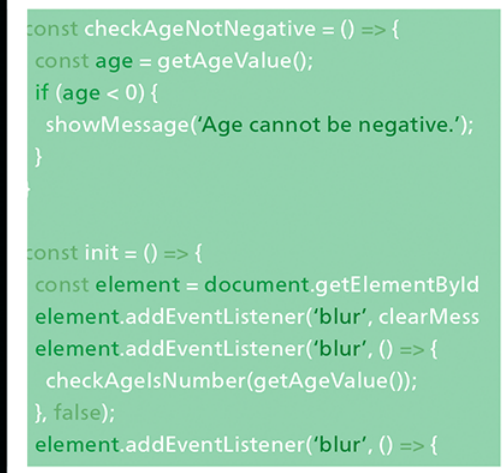
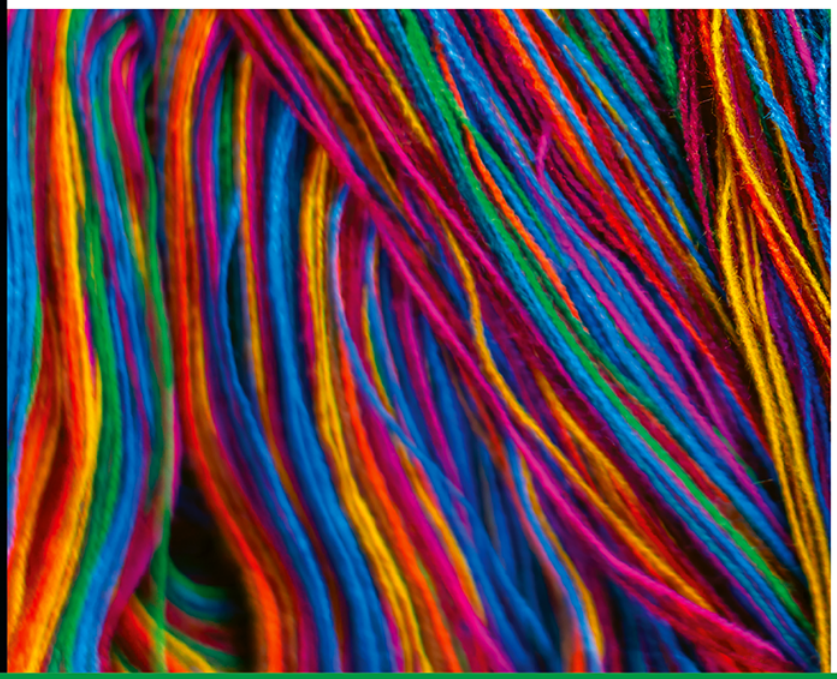
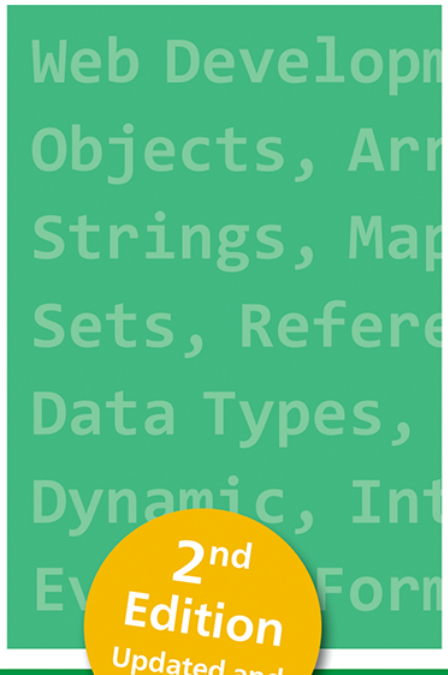






2nd Edition
Updated and Revised

JavaScript

The Comprehensive Guide

Philip Ackermann

Rheinwerk Computing

Contents

Chapter 6
Processing and Triggering Events

An important concept in JavaScript programming is events and thus event-driven programming. In this chapter, we'll show you the principle behind events and how you can use event-driven programming to react to events triggered by users within a (web) application.

When you visit a web page and perform some actions, various *events* are triggered in the background. For example, when the page is fully loaded; when the user enters text into a text field, selects an option from a select box, or clicks on a button; or when elements enter or leave the visible area of the screen. Within your JavaScript code, you have the possibility to react to these events. For example, you could validate the text that the user has entered in a text field or send a form when the user clicks on a button. The principle of sending and responding to events is called *event-driven programming*.

6.1 The Concept of Event-Driven Programming

The concept of event-driven programming is not specific to JavaScript; it's also used in other programming languages. Basically, as shown in also Figure 6.1, on one side, you have triggers for events (also called *event emitters*). In the case of graphical user interfaces (UIs), for example, these can be buttons, text fields, or any other UI components. As soon as you click a button or enter text in a text field, an *event* is triggered in the background by the respective component (the button or the text field).

After an event has been triggered, it's placed in an *event queue*, which ensures that events that were triggered first are also handled first. The *event loop* continuously checks whether there is a new event in the event queue, and if there is, the corresponding event is forwarded to *event handlers*.

In JavaScript, these event handlers are simple functions that allow you, the developer, to respond to the triggered event. So, you can use event-driven programming to express things like "when this button is clicked, this or that function should be called."

However, the principle of event-driven programming isn't limited to interactions with a user interface. In JavaScript development, some events might be triggered without the user interacting with the UI—that is, events that are not triggered by the user but programmatically. In Chapter 17, for example, we'll introduce Node.js, a runtime environment (RTE) for JavaScript that also uses events to control a lot of things in the background. In this chapter,

however, we'll first use web pages (and thus primarily UIs) as an example to show you how to program using events.

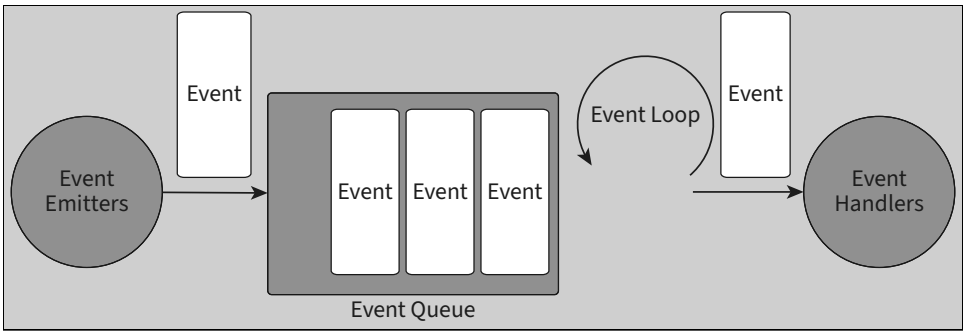


Figure 6.1: The Principle of Events and Event Handling

6.2 Responding to Events

To trigger a function on user interactions within a web page, three steps are required:

1. Select the *element* on the web page that should respond to the user interaction. For example, if you want a function to be triggered when the user clicks on a column heading of a table, you must select the corresponding `<th>` element beforehand.
2. Specify the *event* that should be caught. As you'll see in a moment, there are many different types of events. For example, you can capture mouse clicks, keystrokes, mouse movements, and much more. This step is also called *binding* an event to an element.
3. Specify the *function* to be called. Last but not least, you must define the function to be called when the corresponding event occurs on the selected element. For example, will the data in the table be sorted on click? Will the text entered in a text field be validated?

Implicit Events

Some events in web development can also catch without selecting an element first. These include, for example, events that are triggered when a web page is fully loaded, plus many others.

In the previous chapter, you learned in detail how to select elements on a web page, so step 1 should be a piece of cake for you. So, let's look at how to catch events or bind them to elements and how to define the code to be executed when an event is triggered. In fact, three possibilities in this regard are available:

- **HTML event handlers**
  HTML event handlers provide one possibility to react to events, but they're no longer used nowadays. This option relies on special HTML attributes to define which function

should be called for which event. The reason for not using HTML event handlers anymore is that this option requires mixing HTML code and JavaScript code, which isn't considered a good choice in terms of maintainability and the like. Nevertheless, in Section 6.2.1, we'll describe this type of event handler in detail. You shouldn't use HTML event handlers yourself, but you may come across them in older web applications, and then you should understand what's going on.

- **DOM event handlers**
  The second way to react to events is via what's called a DOM event handler, which are part of the Document Object Model (DOM) specification and are defined via JavaScript. Compared to HTML event handlers, the advantage of using DOM event handlers is that HTML code and JavaScript code are not mixed. In other words, the logic of the application (responding to events) is separate from the structure of the application (the HTML). Nevertheless, DOM event handlers also have a significant downside or limitation: They only allow you to specify a single function to be called for an event. We'll get into the details of this option in Section 6.2.2.
- **DOM event listeners**
  Last but not least, we have a third option to react to events: DOM event listeners. These were introduced in 2000 with the second version of the DOM specification (*www.w3.org/TR/DOM-Level-2-Events/Overview.html*) and have ever since been the standard for event handling and the recommended way to respond to events. The reason: While in the case of event handlers you can only register a single event handler for an event, event listeners allow you to register multiple event listeners for the same event. We'll show you how to define event listeners in Section 6.2.3.

The jQuery Helper Library

In addition, there are various JavaScript libraries that simplify or unify event handling. One of the more prominent examples is the jQuery library, which we'll present in more detail in Chapter 10. A part of Chapter 10 will also deal with event handling.

Definition of Terms

The terms *event handler* and *event listener* are often used synonymously in literature. However, in the case of JavaScript (as described earlier), a distinction is made between the two terms. That is, a function that has been defined as an HTML event handler or a DOM event handler will be referred to as an event handler ahead, and a function that has been defined as an event listener will accordingly be referred to as an event listener.

The essential sequence when an event occurs is shown in Figure 6.2. First, there is a check for whether an event handler or an event listener is defined for the event.

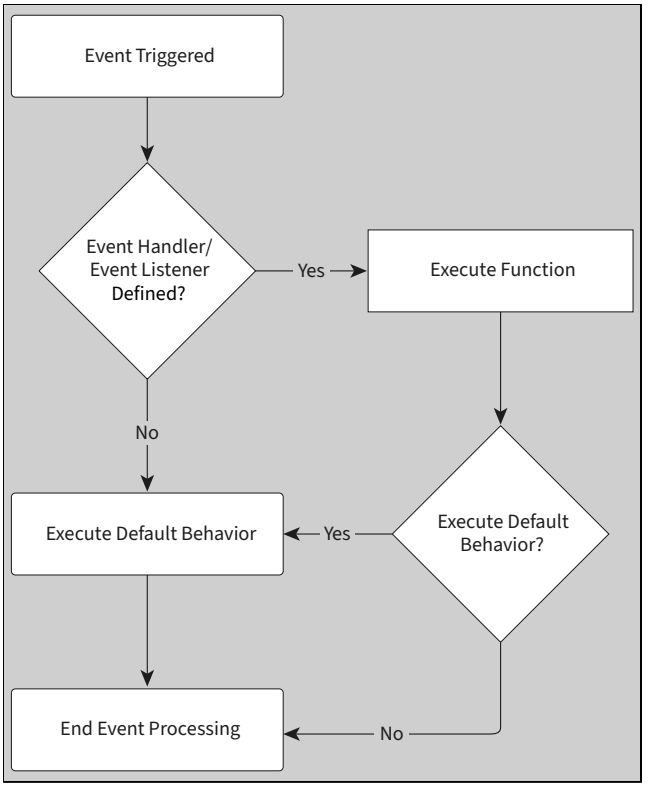


Figure 6.2: Event Handling Procedure

If this is the case, the corresponding code is executed. If, on the other hand, no event handlers/event listeners have been defined, the default actions of the browser that are associated with the respective event are triggered instead. Within an event handler/event listener, you also can disable these default browser actions for an event. We'll show you how in Section 6.4.

6.2.1 Defining an Event Handler via HTML

Let's take a simple use case as an example of event handling: the validation of a form field that should only accept positive numbers. For starters, the only thing to check is that no negative value has been specified. Validation should be triggered every time the user quits the corresponding form field (i.e., when the user clicks outside of the form field).

Native Validation in HTML5

A few years ago, the validation of form fields was only possible using JavaScript. But HTML now includes certain options for checking form fields. However, this feature isn't as flexible as JavaScript, so it's still recommended that web developers know how to use JavaScript to validate the data entered by users on a web page.

Client-Side and Server-Side Validation

Note that, in a production application where users can fill out and submit forms, you should validate the entered data not only on the client side, but also on the server side. Essentially, the client-side validation is conducted for user-friendliness and to alert the user of any errors. The server-side validation, however, ensures that the transmitted data hasn't been (maliciously) manipulated by the user.

Now let's show you how to implement this use case with an HTML event handler. As mentioned earlier, HTML event handlers are defined within the HTML code using HTML attributes. HTML attributes always carry the name of the event preceded by `on`. For the example, the HTML attribute is `onblur` because the `blur` event is triggered when a form element is quit.

The value for the attribute is the function that should be called when the event is triggered. If the function is called `checkAgeNotNegative`, for example, the result is the following HTML attribute:

```
onblur="checkAgeNotNegative()"
```

The full HTML code is shown in Listing 6.1. The input field with the `age` ID is used for entering the age, and the `<div>` element with the `output` ID serves as a container for validation messages.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>HTML event handler</title>
  <link rel="stylesheet" href="styles/main.css">
  <script src="scripts/main.js"></script>
</head>
<body>
  <div>
    <label for="age">Alter: </label>
    <input id="age" type="number" value="0" onblur=
"checkAgeNotNegative()"/>
  </div>
  <div id="output"></div>
</body>
</html>
```

Listing 6.1: Defining an HTML Event Handler Called When an Input Field Is Exited

The called function (i.e., the event handler) is shown in Listing 6.2. Actually, nothing is happening here that we didn't already cover in the previous chapter: First, the input field and

the container are determined by their IDs, and each is assigned to a variable. The value of the input field is then read and checked. If the value is less than `0` (and thus invalid), a corresponding error message is written to the container. On the other hand, if the value is `0` or greater (and thus valid), the text content of the container is cleared (i.e., assigned an empty string). The latter is necessary to clear any error messages if a valid value is entered into the input field after an invalid value.

```
function checkAgeNotNegative() {
  const output = document.getElementById('output');  // Container for
                                                     // message
  const element = document.getElementById('age');    // Input field
                                                     // for age
  const age = element.value;                 // Current age value
  if(age < 0) {                              // If value is negative ...
    output.textContent = 'Age cannot be negative.';  // ... output
                                                     // an error ...
                                                 // ... message...
  } else {                                   // ... else ...
    output.textContent = '';                 // ... delete message.
  }
}
```

Listing 6.2: The Function To Be Called by the Event Handler

If you don't want to trigger the validation as soon as the input field is exited, but only at the push of a button, for example, the code can be adapted relatively easily, as shown in Listing 6.3. In this case, the event handler for the `blur` event has been removed from the text field, and an additional button has been inserted for which the event handler for the `click` event has been defined (which is triggered when the corresponding button is clicked).

```
<input id="age" type="number" value="0"/>
<input type="submit" value="Check age" onclick="checkAgeNotNegative()"/>
```

Listing 6.3: Defining an HTML Event Handler Called When a Button Is Clicked

Although HTML event handlers seem relatively simple to use at first glance, they are now considered obsolete because they mix HTML code and JavaScript code, which can quickly become confusing in large web applications. Instead, you're better off using DOM event handlers, which we'll describe in detail next, or (even better) using event listeners, which we'll present in Section 6.2.3.

6.2.2 Defining an Event Handler via JavaScript

DOM event handlers solve the problem of mixing HTML and JavaScript code. As shown in Listing 6.4, the HTML code remains free of JavaScript calls. Which element should result in which function being called for which event is defined in the JavaScript code.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>HTML event handler</title>
  <link rel="stylesheet" href="styles/main.css">
  <script src="scripts/main.js"></script>
</head>
<body>
  <div>
    <label for="age"></label>
    <input id="age" type="number" value="0"/>
  </div>
  <div id="output"></div>
</body>
</html>
```

Listing 6.4: Using DOM Event Handlers, the HTML Remains Free of JavaScript Code

How this function works is shown in Listing 6.5. First, the input field for which the event handler should be defined is again determined based on the ID. For the `onblur` property of this object, you define the name of the function to be called. Note that, unlike with HTML event handlers, no function is called, so you really just need the name of the function (i.e., without the pair of parentheses). If you were to write `element.onblur = checkAgeNotNegative()` instead of `element.onblur = checkAgeNotNegative`, the return value of the `checkAgeNotNegative()` function call would be defined as an event handler, which is not what we want.

```
function init() {
  const element = document.getElementById('age'); // (1) Get element
  element.onblur =                                 // (2) Define event
    checkAgeNotNegative;                           // (3) Define event
                                                   //     handler
}

window.onload = init;
```

Listing 6.5: Defining a DOM Event Handler

Event Listeners for Loading a Web Page

In Listing 6.5, note that the `onload` property of the `window` object defines an event handler that's called when the web page is loaded. Without this event handler, the `init()` function would not be called, and thus no event handler would be registered on the form element.

DOM event handlers have a small disadvantage, as mentioned earlier: Only one event handler can be defined per event on an element. For example, let's say you want to extend the previous validation to call not only a function that checks whether the entered value is non-negative but also another function that checks whether the entered value is a number at all, which we can call `checkAgeIsNumber()`. As shown in Listing 6.6, the code first defines the `checkAgeNotNegative` function as an event handler for the `onblur` property and then the `checkAgeIsNumber` function. The latter, however, merely ensures that the `onblur` property is reset, and the `checkAgeNotNegative` value is overwritten. The result is that only the `checkAgeIsNumber` function is called, but not the `checkAgeIsNotNegative` function.

```
function init() {
  const element = document.getElementById('age');
  element.onblur = checkAgeNotNegative;
  element.onblur = checkAgeIsNumber;  // Here, the event handler
                                      // is overwritten.
}

window.onload = init;
```

Listing 6.6: Only One Event Handler Can Be Defined per Event

At least for this example, the problem can be solved using anonymous functions. As shown in Listing 6.7, the `checkAgeNotNegative` and `checkAgeIsNumber` functions can simply be encapsulated within such an anonymous function, which is then defined as an event handler.

```
function init() {
  const element = document.getElementById('age');
  element.onblur = function() {    // anonymous function ...
    checkAgeNotNegative();         // ... in which both this function...
    checkAgeIsNumber();            // ... and this function ...
                                   // ... are called
  };
}

window.onload = init;
```

Listing 6.7: Anonymous Functions Partly Avoid the Problem but Are Relatively Inflexible

Although this approach works for the example at hand, you should define functions as event listeners rather than event handlers, if possible. We'll show you how in the following section.

6.2.3 Defining Event Listeners

The disadvantage of event handlers mentioned in the previous section doesn't apply to event listeners or to the related `addEventListener()` method, which is available to all

elements on a web page via the DOM API. As parameters, it expects the name of the event to be responded to and the event listener—that is, the function to be called when the event occurs. Optionally, a third parameter can be specified to influence the *event flow* (more details in Section 6.4).

The example shown in Listing 6.8 does the same thing as the code from the previous sections: It calls the `checkAgeNotNegative()` function when quitting the corresponding form field for entering the age. The first parameter is the `blur` value (without `on`; don't use `onblur`); the second parameter is the name of the function, `checkAgeNotNegative()`.

```
function init() {
  const element = document.getElementById('age');    // Get element
  element.addEventListener(   // Register event listener
    'blur',                   // Name of the event
    checkAgeNotNegative,      // Name of the event listener
    false                     // Event flow, details to follow later on
  );
}
document.addEventListener('DOMContentLoaded', init);
```

Listing 6.8: Defining an Event Listener

Event Listener for Loading the Document Content

Although mentioned earlier in this book (for example, in Chapter 3), let's briefly touch on the `DOMContentLoaded` event again at this point. It's triggered whenever the entire DOM tree of a web page has been loaded, and it's perfect for executing initial code on a web page, such as registering event listeners (or event handlers). The code examples in this book, and especially in the download area for it, we use this event in most cases to execute code inside a web page. A simple example is shown in Listing 6.9.

```
function init() {
  console.log('Document loaded');
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.9: Defining an Event Listener Executed When the Web Page Is Loaded

To define an event listener for the `load` event (analogous to `window.onload`), proceed as described in Listing 6.10. The difference between the `DOMContentLoaded` event and the `load` event is that the former is triggered when the full web document (meaning the DOM tree) has been loaded, while the latter is triggered when the full web document plus all external resources, such as JavaScript files, CSS files, and so on, have been loaded. By the way, no comparable property exists for the `window` object for defining an event handler for the `DOMContentLoaded` event. This event can only be caught via event listeners.

```
function init() {
  console.log('Document and all resources loaded');
}

document.addEventListener('load', init);
```

Listing 6.10: Event Listener Executed When the Web Page, Including All Resources, Has Been Loaded

The attachEvent() Method

The `addEventListener()` method isn't supported in older versions of Internet Explorer. Therefore, if you ever come across the `attachEvent()` method in someone else's code, don't be surprised: This proprietary method of Internet Explorer isn't part of any standard and thus normally should not be used. Only to ensure backward compatibility with old versions of Internet Explorer do libraries like jQuery internally use this nonstandard method in some cases (see also Section 6.2.7). Basically, Internet Explorer is now obsolete, and support was discontinued in mid-2022.

6.2.4 Defining Multiple Event Listeners

Using the `addEventListener()` method, you can now easily register several functions for one and the same event (which is possible with event handlers only via the detour of using a new, usually anonymous function).

In the example shown in Listing 6.11, three different functions are registered as event listeners for the `blur` event on the corresponding form field: The `clearMessage()` function ensures that the message is cleared, the `checkAgeNotNegative()` function checks that the age is not negative, and `checkAgeIsNumber()` ensures that the value entered is a number.

```
function checkAgeNotNegative() {
  const element = document.getElementById('age'); // Input field for age
  const age = element.value;                      // Current age value
  if(age < 0) {                               // If value is negative ...
    showMessage('Age cannot be negative.');  // ... output warning.
  }
}

function checkAgeIsNumber() {
  const element = document.getElementById('age'); // Input field for age
  const age = element.value;                      // Current age value
  if(!(!isNaN(parseFloat(age)) && isFinite(age))) {  // If value is not
                                                  // a number ...
    showMessage('Age must be a number.');         // ... output message.
```

```
  }
}

function clearMessage() {
  showMessage('');
}

function showMessage(message) {
  const output = document.getElementById('output');
  output.textContent = message;
}

function init() {
  const element = document.getElementById('age'); // Get element
  element.addEventListener(                // Register event listener
    'blur',                                // Name of the event
    clearMessage                           // Name of the event listener
  );
  element.addEventListener(                // Register event listener
    'blur',                                // Name of the event
    checkAgeNotNegative                    // Name of the event listener
  );
  element.addEventListener(                // Register event listener
    'blur',                                // Name of the event
    checkAgeIsNumber                       // Name of the event listener
  );
}

document.addEventListener("DOMContentLoaded", init);
```

Listing 6.11: Defining Multiple Event Listeners

Note

The use of event listeners or the `addEventListener()` method is much more flexible compared to event handlers because you can specify not only a single function, but any number of functions.

6.2.5 Passing Arguments to Event Listeners

Sometimes, you may want to pass arguments to the function you have registered as an event listener. However, out of the gate, you can't because you only pass the function itself to the `addEventListener()` method. Again, the solution is using anonymous functions, as shown in Listing 6.12.

The `checkAgeNotNegative()` and `checkAgeIsNumber()` functions now no longer access the form field with the age but expect a numeric value as a parameter to check. As a result, both functions are significantly more reusable because they no longer access the DOM tree directly and could in principle be used in a different context as well.

In addition, the code for accessing the DOM tree or the form field has been moved to the new `getAgeValue()` function. Within the event listeners (anonymous functions in both cases), the value returned by `getAgeValue()` is then passed to the two check methods.

```
function checkAgeNotNegative(age) {
  if(age < 0) {
    showMessage('Age cannot be negative.');
  }
}

function checkAgeIsNumber(age) {
  if(!(!isNaN(parseFloat(age)) && isFinite(age))) {
    showMessage('Age must be a number.');
  }
}

function clearMessage() {
  showMessage('');
}

function showMessage(message) {
  const output = document.getElementById('output');
  output.textContent = message;
}

function getAgeValue() {
  const element = document.getElementById('age');
  const age = element.value;
  return age;
}

function init() {
  const element = document.getElementById('age');
  element.addEventListener(
    'blur',
    clearMessage
  );
  element.addEventListener(
    'blur',
```

```
    function() {                        // anonymous function
      const age = getAgeValue();        // get value for age
      checkAgeNotNegative(age);         // call the actual function
    }
  );
  element.addEventListener(
    'blur',
    function() {                        // anonymous function
      const age = getAgeValue();        // get value for age
      checkAgeIsNumber(age);            // call the actual function
    }
  );
}
document.addEventListener("DOMContentLoaded", init);
```

Listing 6.12: Passing Parameters to Event Listeners

Event Listeners as Arrow Functions

Nothing says you shouldn't define event listeners as arrow functions. The preceding example can therefore be rephrased into the code shown in Listing 6.13.

```
element.addEventListener(
  'blur',
  () => {                               // anonymous arrow function
    const age = getAgeValue();          // get value for age
    checkAgeNotNegative(age);           // call the actual function
  }
);
element.addEventListener(
  'blur',
  () => {                               // anonymous arrow function
    const age = getAgeValue();          // get value for age
    checkAgeIsNumber(age);              // call the actual function
  }
);
document.addEventListener("DOMContentLoaded", init);
```

Listing 6.13: Event Listeners Defined as Arrow Functions

6.2.6 Removing Event Listeners

Event listeners that have been added to an element for an event can also be removed again using the `removeEventListener()` method. As parameters, you pass the name of the event and the event listener to be removed.

An example is shown in Listing 6.14. The age validation example from before has been extended here with a checkbox that can deactivate or reactivate the validation. To this end, the checkbox is assigned an event listener for the `change` event (which triggers whenever the checkbox is enabled or disabled), and within this event listener, the `checked` property determines whether the checkbox is enabled or disabled. In the former case, the `checkAgeNotNegative()` and `checkAgeIsNumber()` event listeners are added, and in the latter case, they are removed via the `removeEventListener()` method.

```
/* Here are the other functions. */
function init() {
  const ageInput = document.getElementById('age');
  ageInput.addEventListener('blur', clearMessage);
  ageInput.addEventListener('blur', checkAgeNotNegative);
  ageInput.addEventListener('blur', checkAgeIsNumber);
  const checkBox = document.getElementById('validation');
  checkBox.addEventListener('change', () => {
    if(checkBox.checked) {
      ageInput.addEventListener('blur', checkAgeNotNegative);
      ageInput.addEventListener('blur', checkAgeIsNumber);
    } else {
      clearMessage();
      ageInput.removeEventListener('blur', checkAgeNotNegative);
      ageInput.removeEventListener('blur', checkAgeIsNumber);
    }
  });
}
document.addEventListener("DOMContentLoaded", init);
```

Listing 6.14: Removing Event Listeners

6.2.7 Defining Event Handlers and Event Listeners via a Helper Function

To support as many browsers as possible without relying on a library like jQuery, you can use the `addEvent()` helper function shown in Listing 6.15. (Remember: Internet Explorer's proprietary method is `attachEvent()`.) The method expects three parameters: the element to which to add the event listener (or event handler), the event to respond to, and the function to register.

The function first checks if the `addEventListener()` method exists (usually checked on the `window` object). If the method does exist, the function is registered as an event listener using this method. If, on the other hand, the method doesn't exist, the function checks in the same way whether the `attachEvent()` method exists; if so, the function is registered via that method. However, if the `attachEvent()` method doesn't exist either, the function is registered as an event handler.

```
function addEvent(element, eventType, eventListener) {
  if (window.addEventListener) {
    element.addEventListener(eventType, eventListener, false);
  }
  else if (window.attachEvent) {
    element.attachEvent('on' + eventType, eventListener);
  }
  else {
    element['on' + eventType] = eventListener;
  }
}
```

Listing 6.15: Helper Function for Defining Event Listeners or Event Handlers

The usage of the `addEvent()` helper function is shown in Listing 6.16.

```
function init() {
  const element = document.getElementById('age');
  addEvent(
    element,
    'blur',
    clearMessage
  );
  addEvent(
    element,
    'blur',
    function() {
      const age = getAgeValue();
      checkAgeNotNegative(age);
    }
  );
  addEvent(
    element,
    'blur',
    function() {
      const age = getAgeValue();
      checkAgeIsNumber(age);
    }
  );
}
```

Listing 6.16: Usage of the Helper Function

6.2.8 Accessing Information of an Event

Within a function that has been registered as an event handler or as an event listener, you can access certain information of the triggered event via a parameter that is passed when calling the corresponding function (but which we omitted in the previous listings for didactic reasons). Each event is represented by a specific object type—for example, mouse events by the `MouseEvent` object type and keyboard events by the `KeyboardEvent` object type. All of these object types have a common (super)type from which they derive: the `Event` type. The properties provided by this type are listed in Table 6.1. (The individual object types like `MouseEvent` and `KeyboardEvent` still provide other properties, some of which we'll discuss later in Section 6.3.)

Property	Description
`bubbles`	Contains an indication of whether or not an event moves upward in the DOM tree (Section 6.4).
`cancelable`	Contains an indication of whether an event can be canceled or not (Section 6.4.2).
`currentTarget`	Contains a reference to the current target of the event (Section 6.4).
`defaultPrevented`	Contains an indication of whether or not the `preventDefault()` method has been called on the event (Section 6.4.3).
`eventPhase`	Contains a numeric value representing the phase the event is currently in (Section 6.4.1). Possible values: ■ 0 (or `Event.NONE`) ■ 1 (or `Event.CAPTURING_PHASE`) ■ 2 (or `Event.AT_TARGET`) ■ 3 (or `Event.BUBBLING_PHASE`)
`target`	Contains a reference to the original target of the event (Section 6.4).
`timeStamp`	Contains the time when the event was triggered.
`type`	Contains the name of the event.
`isTrusted`	Indicates whether the event was triggered by the browser (e.g., by clicking a button) or by JavaScript code (Section 6.5).

Table 6.1: The Properties of Event

A code example for accessing such an event object can be found in Listing 6.17. In this example, a function is registered as an event listener for the `click` event of a button. When the function is called, an object of the `MouseEvent` type is passed to it as a parameter

(which is thus implicitly of the `Event` type as well). Within the function, the various properties of this object are then output (although we've omitted the properties defined by the `MouseEvent` type at this point).

```
function buttonClicked(event) {
  console.log(event.bubbles);           // true
  console.log(event.cancelable);        // true
  console.log(event.currentTarget);     // <input>
  console.log(event.defaultPrevented);  // false
  console.log(event.eventPhase);        // 2
  console.log(event.target);            // <input>
  console.log(event.timeStamp);         // e.g., 1746124230000
  console.log(event.type);              // "click"
  console.log(event.isTrusted);         // true
}

function init() {
  const element = document.getElementById('button'); // Get button
  element.addEventListener(    // Register event listener
    'click',                   // Name of the event
    buttonClicked,             // Name of the event listener
    false                      // Event flow, details to follow later on
  );
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.17: Access to the Event Object

The Event Object in Internet Explorer

As mentioned earlier, older versions of Internet Explorer do not always behave in a standards-compliant manner, which is also true with regard to the event object. In these browsers, this object isn't passed as an argument to the event handler (event listeners don't exist in these older browser versions) but is provided as an `event` property of the global `window` object (this property always contains only the currently triggered event).

Fortunately, the event object is still essentially structured as described in Table 6.1. However, the target element of the event (usually stored in the `target` property) must be accessed via the `srcElement` property.

You can access the event object and the target element in a browser-independent way as shown in Listing 6.18. (In the following examples in this book, however, we won't use this browser-independent solution for the sake of clarity.)

```
function handler(ev) {
  var e = ev || window.event;               // Get event
  var target = e.target || e.srcElement;    // Target element
}
```

Listing 6.18: Browser-Independent Access to the Event Object and the Target Element

Recall that the type of the event object differs depending on the type of the triggered event. The available types of events—and thus also the available object types—will be detailed in the following sections.

6.3 The Different Types of Events

Within a web application, different types of events might occur. Essentially, the following types can be distinguished:

- Events when interacting with the mouse (Section 6.3.1)
- Events when interacting with the keyboard or with text fields (Section 6.3.2)
- Events when working with forms (Section 6.3.3)
- Events when focusing elements (Section 6.3.4)
- Events related to the UI (Section 6.3.5)
- Events on mobile devices (Section 6.3.6)

Other Events

In addition to the events we'll present in this chapter, there are a number of others you'll encounter when developing with JavaScript. Some of these are discussed later in the book:

- In Chapter 9, we'll explain how to dynamically reload web page content from the server. This is another use case for events.
- In Chapter 10, we'll introduce the jQuery library, which simplifies and unifies many aspects of JavaScript development, including how events are handled.
- In Chapter 11, we'll address the topic of creating images using JavaScript, and that chapter also uses events.
- Also, the various web APIs outlined in Chapter 12 provide various events that you can respond to as a web developer.
- Node.js, the subject of Chapter 17, also uses the principle of event-driven programming.

In addition, you can find a good overview of the various events at *https://developer.mozilla.org/en-US/docs/Web/Events*.

6.3.1 Events When Interacting with the Mouse

The classic form of interaction with web pages is by using the mouse. Whenever a user moves the mouse or presses the mouse buttons, an event is triggered in the background. The related events are shown in Table 6.2. In addition to simple mouse clicks (`click` event), there is an event for double-clicking (`dblclick`) and for when the context menu has been opened (`contextMenu`).

The `mousedown` and `mouseup` events are triggered when the mouse button is pressed and released, respectively, and the `mousemove` event is triggered whenever the mouse is moved. In addition, there are various events that are triggered depending on whether the mouse pointer is moved over or away from an element on the web page.

Event	Description	Object Type
`click`	Triggered when the mouse button is pressed and released over an element	`MouseEvent`
`dblclick`	Triggered when the mouse button is pressed and released twice over an element	`MouseEvent`
`contextmenu`	Triggered when a context menu has been opened (usually via the right mouse button)	`MouseEvent`
`mousedown`	Triggered when the mouse button is pressed over an element	`MouseEvent`
`mouseup`	Triggered when the mouse button is released over an element	`MouseEvent`
`mousemove`	Triggered when the mouse has been moved	`MouseEvent`
`mouseover`	Triggered when the mouse is moved over an element, as shown in also note box	`MouseEvent`
`mouseout`	Triggered when the mouse is moved away from an element, as shown in also note box	`MouseEvent`
`mouseenter`	Triggered when the mouse is moved over an element or the mouse pointer enters the element's area, as shown in also note box	`MouseEvent`
`mouseleave`	Triggered when the mouse is moved away from an element, as shown in also note box	`MouseEvent`

Table 6.2: Overview of Events When Interacting with the Mouse

mouseenter/mouseleave versus mouseover/mouseout

The `mouseenter` and `mouseover` events and the `mouseout` and `mouseleave` events are quite similar at first glance. The former two are triggered when the mouse is moved over an element, the latter when the mouse is moved away from an element. The exact difference is shown in Figure 6.3: `mouseenter` and `mouseleave` are only triggered when the outer edge of the respective element is crossed, while `mouseover` and `mouseout` are additionally triggered when another element lies within the respective element and the mouse is moved over it (i.e., moved away from the outer element) or moved away from the inner element (and moved over the outer element).

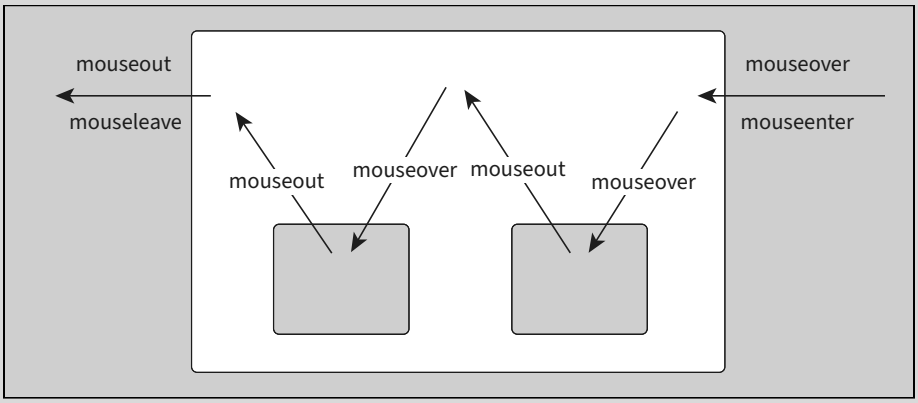


Figure 6.3: The mouseenter, mouseout, mouseover, and mouseleave Mouse Events in Comparison

Plus, you'll notice that all mouse events are of the `MouseEvent` object type. Besides the properties defined by the (super)type `Event`, this type provides the properties shown in Table 6.3.

Property	Description
`altKey`	Contains the Boolean indicating whether the `Alt` key was pressed when the event was triggered.
`button`	Contains the number of the mouse button that was clicked when the event was triggered.
`buttons`	Contains a number representing the mouse buttons clicked when the event was triggered. The number 1 represents the left mouse button, the number 2 the right mouse button, the number 4 the mouse wheel or the middle mouse button, the number 8 the fourth mouse button, and the number 16 the fifth mouse button. If several buttons were pressed when the event was triggered, `buttons` contains the sum of the respective numbers.

Table 6.3: Overview of the Properties of the MouseEvent Object Type

Property	Description
`clientX`	The *x*-coordinate (relative to the DOM content) where the mouse pointer is located when the event is triggered.
`clientY`	The *y*-coordinate (relative to the DOM content) where the mouse pointer is located when the event is triggered.
`ctrlKey`	Contains the Boolean indicating whether the `Ctrl` key was pressed when the event was triggered.
`metaKey`	Contains the Boolean indicating whether the *meta key* was pressed when the event was triggered, that is, for macOS, the `Cmd` key; for Windows, the `Windows` key.
`movementX`	The *x*-coordinate relative to the previous *x*-coordinate that occurred when the last `mousemove` event was triggered.
`movementY`	The *y*-coordinate relative to the previous *y*-coordinate that occurred when the last `mousemove` event was triggered.
`region`	Contains the ID of the region (or element) to which the event refers.
`relatedTarget`	Contains the related target element that is thematically connected to the current target element by the event. For example, when the mouse pointer is dragged away from one element and onto another, a `mouseenter` event is triggered for the latter. The `relatedTarget` property then refers to the element from which the mouse pointer was dragged away.
`screenX`	The *x*-coordinate (relative to the screen) where the mouse pointer is located when the event is triggered.
`screenY`	The *y*-coordinate (relative to the screen) where the mouse pointer is located when the event is triggered.
`shiftKey`	Contains the Boolean indicating whether the `Shift` key was pressed when the event was triggered.

Table 6.3: Overview of the Properties of the MouseEvent Object Type (Cont.)

Note

For reasons of space, we won't go into all properties in detail for the following object types of events, and we won't include the corresponding overview tables either. If you're specifically looking for the details of a particular object type or the various APIs in general, a good place to go is the web page mentioned earlier: *https://developer.mozilla.org/en-US/docs/Web/API/Event*.

An example of responding to mouse events, which is a bit ahead in terms of content but is quite illustrative and offers something to play around with, is shown in the following two listings. This example illustrates how a part of the browser window can be used as a drawing area using the Canvas API (see also Chapter 11) and the `mousemove` event.

Listing 6.19 shows the necessary HTML code to serve as a basis. The highlighted `<canvas>` element represents the drawing area (more on this topic in Chapter 11). Listing 6.20 shows the JavaScript code that ensures that, when the mouse moves over the drawing area, corresponding lines are drawn along the mouse movements.

The `handleMouseMove()` function is registered at the drawing area as an event listener for the `mousemove` event. As soon as you move the mouse pointer over this drawing area, this event is triggered in the background, and the `handleMouseMove()` function is executed. Within this function, the `clientX` and `clientY` properties of the event object are then accessed, which are specific to the `MouseEvent` object type. These two properties contain the *x* and *y* coordinates of the mouse pointer at the time the event was triggered. Based on this position information, the next part of the function will then accordingly draw the path on the drawing area. (You don't need to understand this code in detail now; it contains much that we won't discuss until Chapter 11.)

```
<!DOCTYPE html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Draw based on mouse position</title>
  <link rel="stylesheet" href="styles/main.css">
  <script src="scripts/main.js"></script>
</head>
<body>
<div id="container">
  <canvas id="canvas" width="400" height="400">
  </canvas>
</div>
</body>
</html>
```

Listing 6.19: Source HTML with Drawing Area

```
function init() {
  const canvas = document.getElementById('canvas');
  canvas.addEventListener('mousemove', handleMouseMove, false);
  const context = canvas.getContext('2d');  // Get drawing area.
  let started = false;                      // Notice start of path.
  function handleMouseMove(event) {
    let x, y;
    if (event.clientX           // If x position is specified ...
      || event.clientX == 0) {  // ... and not 0 ...
```

```
      x = event.clientX;        // ... remember x position ...
      y = event.clientY;        // ... remember y position.
    }
    if (!started) {             // If path not yet started ...
      context.beginPath();      // ... start path and ...
      context.moveTo(x, y);     // ... move to position.
      started = true;           // Notice that path has started.
    } else {                    // If path has started ...
      context.lineTo(x, y);     // ... move to position ...
      context.stroke();         // ... and draw connection.
    }
  }
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.20: Drawing on the Canvas Using the Canvas API and the mousemove Event

6.3.2 Events During Interactions with the Keyboard

Events are also triggered when the keyboard is used. Each time a user presses a key on the keyboard, three events are triggered: the `keydown` event when the respective key is pressed down; `keyup` when the key is released; and (in between) `keypress`, which virtually represents the typed character. The events that are triggered when interacting with the keyboard are of the `KeyboardEvent` type, as shown in Table 6.4.

Event	Description	Object Type
`keydown`	Triggered when a key is pressed. If a key is pressed for a longer time, the event is triggered several times.	`KeyboardEvent`
`keyup`	Triggered when a key is released.	`KeyboardEvent`
`keypress`	Triggered when a character is inserted via the keyboard. The following principle applies: If a key is pressed for a longer time, the event is triggered several times. However, this event is now outdated.	`KeyboardEvent`

Table 6.4: Overview of Events When Interacting with the Keyboard

For an example of using keyboard events, see the relatively extensive (but again, very practical) Listing 6.21, which shows how to move a web page element (in this example, a circular `<div>` element) using the arrow keys.

For this purpose, the `keydown` event is caught by means of an event listener. Within the event listener, the pressed key is determined, and depending on this information, one of

four methods is called: `moveUp()`, `moveDown()`, `moveRight()`, or `moveLeft()`. Each of these methods updates the position of the element stored in the `position` array and calls the `move()` method, which in turn updates the position of the element via CSS.

```
function init() {
  window.addEventListener('keydown', (event) => {
      console.log(event.key);
      switch (event.key) {
        case 'ArrowUp':
          moveUp();
          break;
        case 'ArrowLeft':
          moveLeft();
          break;
        case 'ArrowRight':
          moveRight();
          break;
        case 'ArrowDown':
          moveDown();
          break;
        default: return;
    }
  });

  const circle = document.getElementById('circle');
  const position = [0, 0];
  const offset = 10;

  function move() {
    circle.style.top = position[0] + 'px';
    circle.style.left = position[1] + 'px';
  }
  function moveUp() {
    position[0] -= offset;
    move();
  }

  function moveRight() {
    position[1] += offset;
    move();
  }

  function moveLeft() {
    position[1] -= offset;
```

```
    move();
  }

  function moveDown() {
    position[0] += offset;
    move();
  }
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.21: Moving an Element Using the Arrow Keys and the keydown Event

6.3.3 Events When Working with Forms

Special events exist for working with forms. For example, corresponding events are triggered whenever the user modifies the text in an `<input>` or a `<textarea>` element; whenever the selected value in a dropdown list, checkbox, a group of radio buttons changes; or when a form is submitted or the values contained in a form are reset. All these events are represented by the `Event` object type, so they don't have their own specific object type like mouse events or keyboard events, as shown in Table 6.5.

Note

We won't include any code examples in this section because we'll cover forms in more detail in Chapter 7, where you'll also learn how to use form events.

Event	Description	Object Type
`input`	Triggered when the value of an `<input>` or a `<textarea>` element or an element with the `contenteditable` attribute has been changed	`Event`
`change`	Triggered when the selected value of a selection list, a checkbox, or a group of radio buttons has been changed	`Event`
`submit`	Triggered when a form has been submitted	`SubmitEvent`
`reset`	Triggered when a form has been reset (via a reset button)	`Event`

Table 6.5: Events When Interacting with Text Fields and Forms

6.3.4 Events When Focusing Elements

In addition, an event is triggered every time an element on a web page goes into focus, for example, by navigating to the element using the Tab key, or when it loses the focus. An

overview is provided in Table 6.6. Note how the object types of the corresponding triggered events are each of the `FocusEvent` type.

Note

Remember that, in the opening example for this chapter, you learned how the `blur` event can be used to validate text inputs in form fields.

Event	Description	Object Type
`focus`	Triggered when an element is focused.	`FocusEvent`
`blur`	Triggered when an element loses focus.	`FocusEvent`
`focusin`	Triggered when an element is focused. Unlike the `focus` event, this event bubbles (Section 6.4).	`FocusEvent`
`focusout`	Triggered when an element loses focus. Unlike the `blur` event, this event bubbles (Section 6.4).	`FocusEvent`

Table 6.6: Overview of Events When Focusing Elements

6.3.5 General Events of the User Interface

In addition to the events we've presented so far, a number of other events can be roughly categorized as UI events, where most, but not all, of these events are represented by the `UIEvent` type. An overview is presented in Table 6.7.

Event	Description	Object Type
`load`	Triggered when a web page is fully loaded	`UIEvent`
`unload`	Triggered when a web page is unloaded, usually when a new web page is requested in the corresponding browser window	`UIEvent`
`abort`	Triggered when the loading of a resource is canceled	`UIEvent`
`Error`	Triggered when there is a problem loading the web page, such as a JavaScript error	`UIEvent`
`select`	Triggered when text is selected on a web page	`UIEvent`

Table 6.7: Events Related to the User Interface

Event	Description	Object Type
`resize`	Triggered when the size of the browser window has been changed	`UIEvent`
`scroll`	Triggered when scrolling up or down	`UIEvent`
`beforeunload`	Triggered just before a web page is unloaded, usually when a new web page is requested in the corresponding browser window	`BeforeUnloadEvent`
`DOMContentLoaded`	Triggered when the DOM tree is fully loaded	`Event`
`cut`	Triggered when content has been cut from a form field, for example, from pressing the keyboard shortcut Cmd + X	`ClipboardEvent`
`copy`	Triggered when content has been copied from a form field, for example, from pressing the keyboard shortcut Cmd + C	`ClipboardEvent`
`paste`	Triggered when content has been inserted into a form field, for example, from pressing the keyboard shortcut Cmd + V	`ClipboardEvent`
`select`	Triggered when text is selected in a form field	`ClipboardEvent`

Table 6.7: Events Related to the User Interface (Cont.)

Usage of the load Event

In the example of using the `load` event earlier in this chapter, we talked about executing initial JavaScript code (and registering further event handlers or event listeners) after the web page has loaded.

One use case for the `scroll` event, for example, is to display elements of a web page depending on the vertical scroll position. You've probably seen this effect before, now often used for what's called single-page applications (SPAs) where the entire content is contained on a single, vertical, very long web page. If the user scrolls down on such a web page, further components of the web page gradually fade in, such as text or images (sometimes also moving into the screen from the left or right).

The next two listings show a simple implementation (at least visually) to displays a text depending on the vertical scroll position. Listing 6.22 only contains the HTML code and is

only intended to show you the structure of the web page. (The complete example including CSS can be found, as always, in the download area for the book.)

The `<span>` element with the `name` ID that contains the text `John Doe` is initially hidden when the web page loads. Listing 6.23 also registers an event listener (`handleScrollEvent()`) for the `scroll` event on the `window` object when the web page is loaded. Within this event listener, `window.scrollY` is used to determine the vertical scroll position (see also Chapter 8). When this scroll position exceeds a certain threshold (the value `10` in the example), the `hide` CSS class is removed from the hidden `<span>` element and replaced with the `show` CSS class. The CSS rule defined for this CSS class (not shown here) then ensures that the element is slowly faded in.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Fade in an element when scrolling</title>
  <link rel="stylesheet" href="styles/main.css">
  <script src="scripts/main.js"></script>
</head>
<body>
  <div id="content">
    Hello
    <span id="name" class="hide">John Doe</span>
  </div>
</body>
</html>
```

Listing 6.22: Source HTML for Fading in an Element Depending on the Scroll Position

```
function init() {
  let scrollPosition = window.scrollY;
  const nameElement = document.getElementById('name');

  function handleScrollEvent(e) {
    scrollPosition = window.scrollY;
    if(scrollPosition > 10) {
      nameElement.classList.remove('hide');
      nameElement.classList.add('show');
    }
    else {
      nameElement.classList.remove('show');
      nameElement.classList.add('hide');
    }
  }
  window.addEventListener('scroll', handleScrollEvent);
```

```
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.23: Displaying an Element Depending on the Scroll Position Using an Event Listener for the Scroll Event

6.3.6 Events on Mobile Devices

When using mobile devices (smartphones, tablets, etc.), a number of other events are triggered, primarily because such end devices can be operated in quite a different way than conventional laptops or desktop computers. Instead of using a mouse and keyboard, you usually use your fingers or special input pens to interact with the corresponding website.

Based on these forms of interaction—which are very much designed for touching the display—corresponding events can detect and respond to touches on the display of the end device.

In addition, other factors can play a role with mobile devices, such as the device's tilt or orientation: Is it tilted sideways or tilted forward or backward? Events can respond to the movement of a mobile device. An overview of the most important events in this regard can be found in Table 6.8.

Event	Description	Object Type
`deviceorientation`	Triggered when new data is available regarding the orientation of the end device	`DeviceOrientationEvent`
`devicemotion`	Triggered at regular intervals and then indicates the acceleration force acting on the end device	`DeviceMotionEvent`
`touchstart`	Triggered when the input device (usually a finger) touches the display	`TouchEvent`
`touchend`	Triggered when the input device stops touching—in other words, when the finger is removed from the display	`TouchEvent`
`touchmove`	Triggered when the input device has been moved over the display	`TouchEvent`
`touchcancel`	Triggered when the tracking of the touch has been interrupted	`TouchEvent`

Table 6.8: A Selection of Mobile Device Events

Note

To remain within the scope of this chapter, we won't include code examples here but will provide them in Chapter 12. In that chapter, you'll learn how to recognize and respond to the orientation and movements of a mobile device.

6.4 Understanding and Influencing the Flow of Events

Whenever an event is triggered, it goes through several event phases. As a JavaScript developer, it's important for you to understand what these phases are and how they're related to event triggering.

6.4.1 The Event Phases

Basically, the following three phases are distinguished with regard to the event flow:

1. **Capturing phase**
   In this phase, the event "descends" from the top node in the DOM tree (the document node) down to the element for which the event was triggered (the target element), as shown in Figure 6.4.

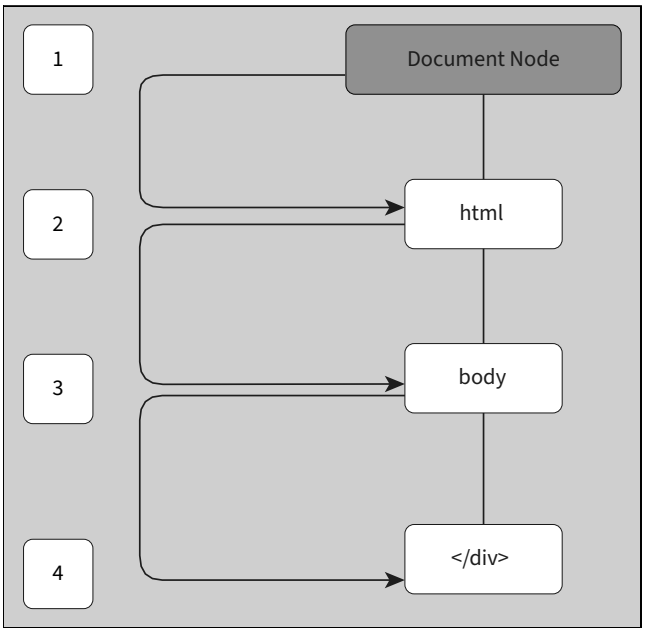


Figure 6.4: In Event Capturing, the Event Descends to the Bottom

2. **Target phase**
   In this phase, the event is triggered at the target element.

3. **Bubbling phase**
   In this phase, the event "rises" from the target element in the DOM tree back up to the document node, as shown in Figure 6.5.

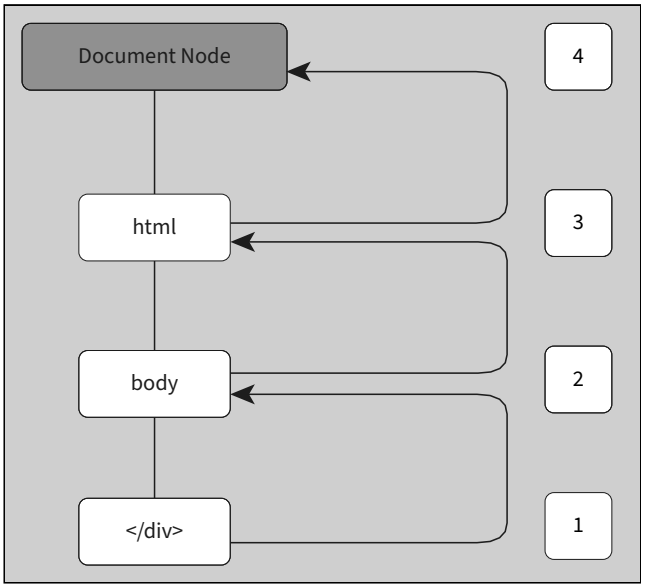


Figure 6.5: In Event Bubbling, the Event Rises to the Top

During the capturing phase and the bubbling phase, the event triggers the event handlers or event listeners that have been registered for this event at the respective element at which the event is currently located.

The capturing phase and the bubbling phase trace back to differing opinions of the various browser manufacturers a long time ago. Browser manufacturer Netscape at the time believed that an event should always first call the event handler/event listener on the element that is further up in the DOM tree and therefore proceeded according to what's called *event capturing*, which consisted only of the capturing phase (and the target phase). But Microsoft held a different opinion: In Internet Explorer, an event first triggers the event handlers/event listeners defined on the target element. In other words, Microsoft followed the *event bubbling* principle, which in turn consisted only of the bubbling phase (and the target phase).

However, the event flow has been standardized by the World Wide Web Consortium (W3C) for some time now, as shown in *www.w3.org/TR/DOM-Level-3-Events/#event-flow*, with care taken to combine both phases into one event flow model, as shown in Figure 6.6. Now, when an event is triggered, it first goes through the capturing phase, executing all event handlers/event listeners registered for that phase (more on that later); then goes through the target phase, executing the event handlers/event listeners registered on the target element; and then goes through the bubbling phase, now executing all event handlers/event listeners registered for that phase.

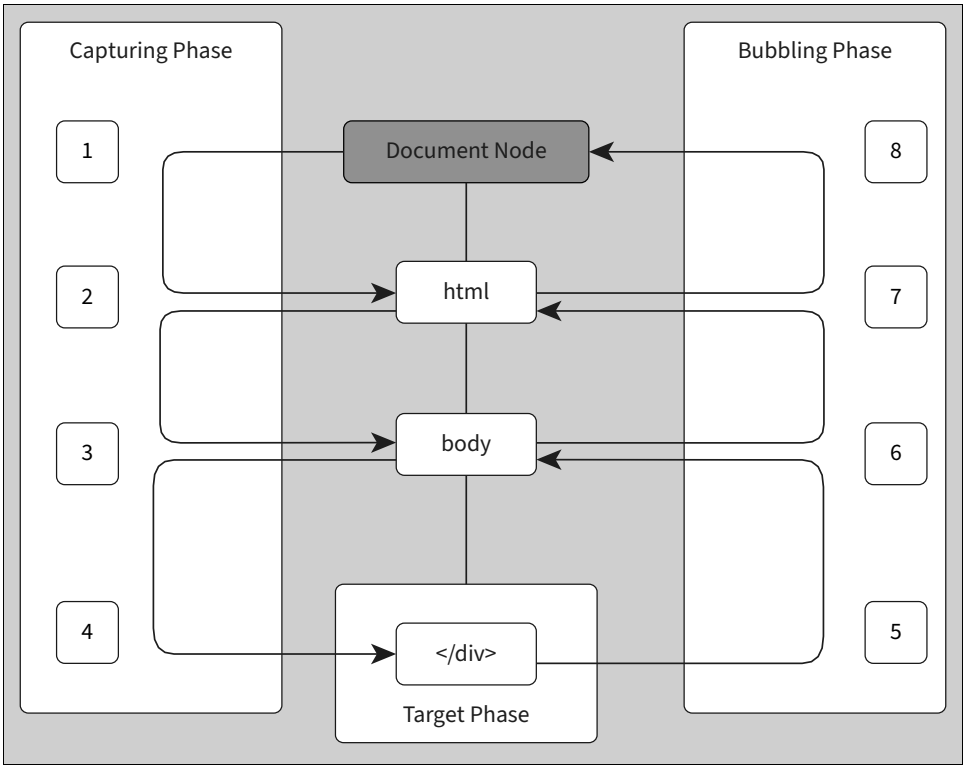


Figure 6.6: The W3C's Event Flow Model Combining the Capturing, Target, and Bubbling Phases

The easiest way to understand this differences between event bubbling and event capturing is to look at some examples.

Event Bubbling Example

Let's start with an example of event bubbling. First, look at the HTML code shown in Listing 6.24, which contains an outer `<div>` element (with the ID `outer`) that contains two `<div>` elements (with the IDs `first` and `second`). These two elements contain two `<div>` elements each. The structure is shown in Figure 6.7.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Event capturing and event bubbling</title>
  <link rel="stylesheet" href="styles/main.css">
  <script src="scripts/main.js"></script>
</head>
<body>
<div id="outer" class="level1">
```

```
  <div id="first" class="level2">
    1
    <div id="A" class="level3">A</div>
    <div id="B" class="level3">B</div>
  </div>
  <div id="second" class="level2">
    2
    <div id="C" class="level3">C</div>
    <div id="D" class="level3">D</div>
  </div>
</div>
</body>
</html>
```

Listing 6.24: The HTML for the Event Bubbling Example

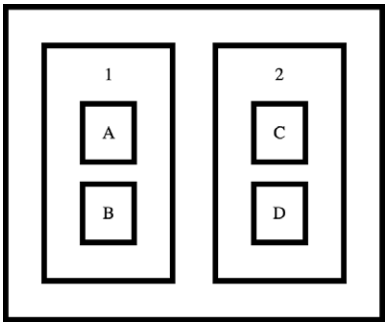


Figure 6.7: The Starting Point for the Event Bubbling Example

In the JavaScript code shown in Listing 6.25, the same function is now added to each of these `<div>` elements as an event listener. This function does nothing but toggle the `selected` CSS class via `classList.toggle()` and thus switch between a white and gray background.

```
function handler(ev) {
  const e = ev || window.event;                    // Get event
  const target = e.target || e.srcElement;         // Target element
  this.classList.toggle('selected');               // CSS class
  console.log(                                     // Output clicked
...
    `Clicked on node with ID "${target.id}"`       // ... element
  );
  console.log(                                     // Output current
...
    `Event at node with ID "${this.id}"`           // ... element
  );
```

```
}
function init() {
  const elements = document.querySelectorAll(   // All elements ...
    '.level1, ' +                                // ... of the first, ...
    '.level2, ' +                                // ... the second, ...
    '.level3'                                    // ... and the third ...
  );                                             // ... level...
  for(let i=0; i<elements.length; i++) {         // ... get a ...
    elements[i].addEventListener(                // ... listener for ...
      'click',                                   // ... the click event.
      handler,
      false
    );
  }
}
document.addEventListener('DOMContentLoaded', init);
```

Listing 6.25: Event Bubbling Example

If you now click on the `<div>` element that contains the `C` (i.e., on an element of the third `<div>` layer, so to speak), this element is colored gray. After that, the event is passed to the parent element (with the `2`), which also colors this element gray. Then the event is forwarded again (to the outermost `<div>` element), the event listener is executed there as well, and accordingly the background color of this element is also set to gray. You can see the result in Figure 6.8, and the output of the program in Listing 6.26.

```
Clicked on node with ID "C"
Event at node with ID "C"
Clicked on node with ID "C"
Event at node with ID "second"
Clicked on node with ID "C"
Event at node with ID "outer"
```

Listing 6.26: Output for the Previous Program When Clicking on the Element Labeled C

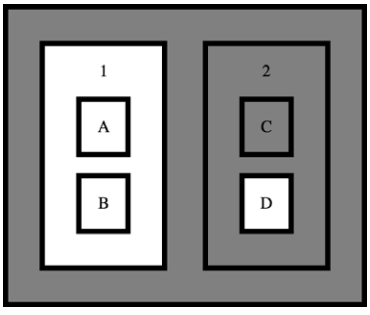


Figure 6.8: Clicking on the Element Labeled C Turns All Preceding Elements Gray During Event Bubbling, Starting from the Target Element

If, on the other hand, you click on the `<div>` element with the `2` (i.e., an element of the second `<div>` layer), its event listener is executed first, and the background color is set to gray. Then, the event listener for the parent element—that is, the outermost `<div>` element—is executed, and its background color is also set to gray. The result is shown in Figure 6.9, and the output of the program, in Listing 6.27.

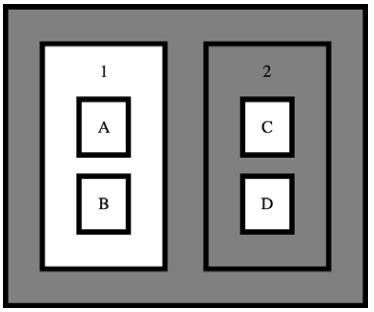


Figure 6.9: Clicking on the Element Labeled 2, Turns All Preceding Elements Gray During Event Bubbling, Starting from the Target Element

```
Clicked on node with ID "second"
Event at node with ID "second"
Clicked on node with ID "second"
Event at node with ID "outer"
```

Listing 6.27: Output for the Preceding Program when Clicking on the Element Labeled 2

Notice how the event listeners that you register via the `addEventListener()` method are registered for the bubbling phase by default. Next, let's look at how to register event listeners for the capturing phase.

Event Handlers in the Bubbling Phase

If you register an event handler (whether via HTML or via JavaScript), it's always executed in the bubbling phase. In contrast to event listeners, event handlers don't offer a way to influence the phase in which they should be executed.

Event Capturing Example

Using the `addEventListener()` method, you can register a function as an event listener for the capturing phase. This method can be controlled via a third parameter: You can specify a Boolean value that determines whether the respective function should be registered as an event listener for the bubbling phase or for the capturing phase.

By default, as already mentioned, the registration takes place for the bubbling phase, but if you want the event listener to be registered for the capturing phase, you need to pass the value `true`. An example of this is shown in Listing 6.28, which is essentially based on the

previous examples in which we demonstrated event bubbling. The only difference is that the event listeners for the `click` event are now registered for the capturing phase rather than the bubbling phase.

```
function init() {
  const elements = document.querySelectorAll('.level1, .level2,
.level3');
  for(let i=0; i<elements.length; i++) {
    elements[i].addEventListener('click', handler, true);
    // elements[i].onclick = handler;           // always bubbling phase
    // elements[i].attachEvent('click', handler);  // IE<9 always
                                                   // bubbling phase
  }
}
```

Listing 6.28: Event Capturing Example

If you now click on the element with the value `C` again, the event won't trigger the event listeners in the bubbling phase starting from the target element, but in the capturing phase starting from the document node (of course, the bubbling phase is still run through in order to be able to execute other event handlers and such event listeners that were registered for the bubbling phase).

The first element that has an event listener registered for that event is the outermost `<div>` element, which means that the first thing to execute is the event listener for that element. Then the event moves down the DOM tree (we're in the capturing phase) and triggers the event listener for the element with the `2`. Finally, the event listener is triggered for the element that was clicked (the target element with the `C`).

The output of the program, shown in Listing 6.29, illustrates this sequence.

```
Clicked on node with ID "C"
Event at node with ID "outer"
Clicked on node with ID "C"
Event at node with ID "second"
Clicked on node with ID "C"
Event at node with ID "C"
```

Listing 6.29: Output of the Preceding Program

The graphical result, in turn, resembles the result from the event bubbling example when the `<div>` element with the `C` was also clicked, as shown in Figure 6.10 (compare with Figure 6.8).

Even if you click on the `<div>` element with the `2`, the graphical result looks the same as in the event bubbling example, as shown in Figure 6.11 (compare with Figure 6.9). Only the order in which the event listeners are executed is different.

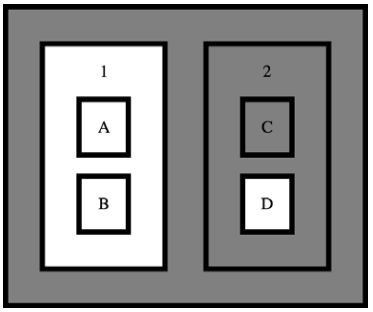


Figure 6.10: If You Click on the C Element, All Preceding Elements Are Colored Gray During Event Capturing, Starting from the Outermost Element

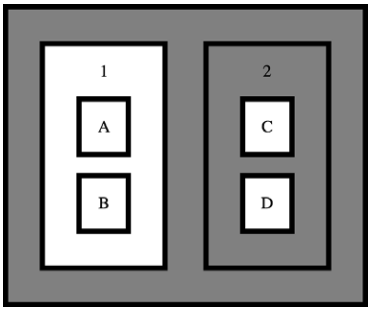


Figure 6.11: Clicking on the Element Labeled 2 Turns All Preceding Elements Gray During Event Capturing, Starting from the Outermost Element

6.4.2 Interrupting the Event Flow

When an event is triggered, actions that normally occur as a result of the event can be prevented:

- The `stopPropagation()` method can be used to prevent the corresponding event from being propagated to the next element in the DOM tree, thereby triggering actions (or corresponding event handlers and event listeners) on other elements.
- The `preventDefault()` method, on the other hand, can be used to prevent the browser's default actions from being executed when an event occurs. For example, you can prevent the browser from opening a link when you click on it or from submitting a form when you click on a submit button.

The code shown in Listing 6.30 is an expansion of the program: An additional `if` statement checks whether the current element (`this`) contains the CSS class `level2`—in other words, whether the current element is an element of the second level, one of the two `<div>` elements with the `1` or the `2` (the outermost `<div>` element has the CSS class `level1`, while the four innermost `<div>` elements have the CSS class `level3`; see also Listing 6.24). If this is the case, the `stopPropagation()` method is executed on the event object.

So if you now click on a third-level element—for example, again on the element containing the C—then by default (with an event listener registered for the bubbling phase) only that element and the element containing the 2 are colored gray, but not the outermost element, as shown in Figure 6.12 with the output of the program shown in Listing 6.31. Because forwarding is prevented via the stopPropagation() method, the outermost element is not reached, as shown in Figure 6.13.

```
function handler(ev) {
  const e = ev || window.event;
  const target = e.target || e.srcElement;
  this.classList.toggle('selected');
  console.log(
    `Clicked on node with ID "${target.id}"`
  );
  console.log(
    `Event at node with ID "${this.id}"`
  );
  if(this.classList.contains('level2')) {  // Once level 2 has been ...
    e.stopPropagation();           // ... reached the event will not ...
  }                                // ... be forwarded.
}
```

Listing 6.30: Termination of the Event Flow

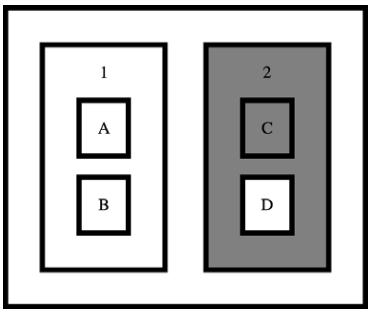


Figure 6.12: When Interrupting the Event Flow in the Bubbling Phase, Only the Target Element up to the 2 Element Is Grayed

```
Clicked on node with ID "C"
main.js:8 Event at node with ID "C"
main.js:5 Clicked on node with ID "C"
main.js:8 Event at node with ID "second"
```

Listing 6.31: Output for the Preceding Program When Clicking on the C Element

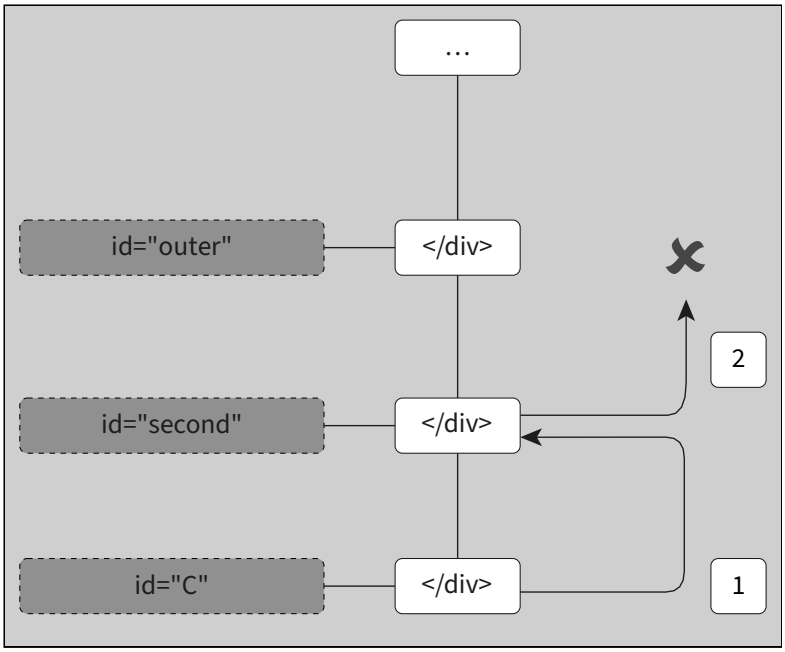


Figure 6.13: Calling stopPropagation() Prevents the Event from Rising in the DOM Tree During the Bubbling Phase

Event Listeners in the Capturing Phase

The example assumes that the event listeners have all been registered for the bubbling phase. If, on the other hand, the event listeners are registered for the *capturing* phase, the result looks different, as shown in Figure 6.14, because the event listeners are evaluated at the respective elements starting from the document node, as shown in Figure 6.15.

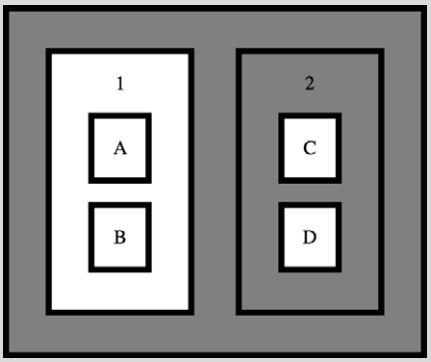


Figure 6.14: When Interrupting the Event Flow in the Capturing Phase, Only the Document Node up to the 2 Element Is Grayed

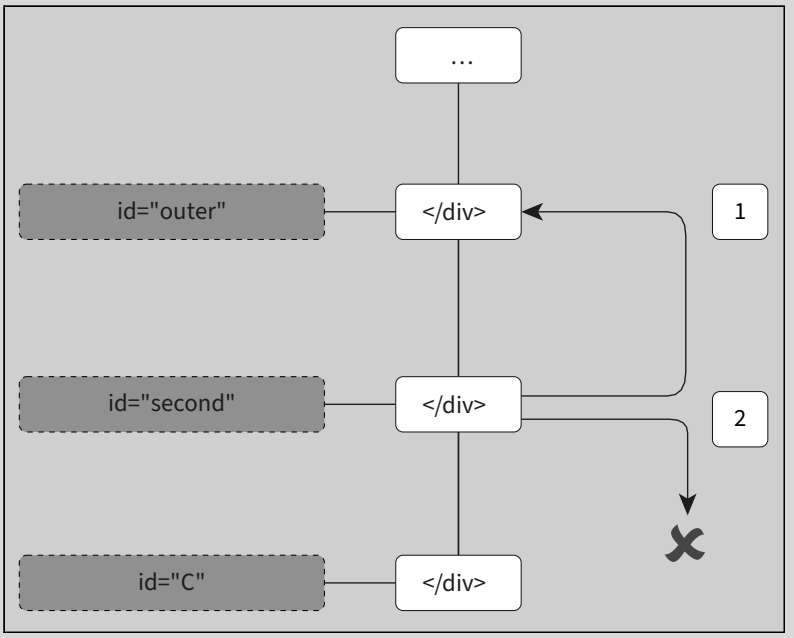


Figure 6.15: Calling stopPropagation() Prevents the Event from Descending in the DOM Tree During the Capturing Phase

Interrupting the Event Flow Immediately

As you know, you can register more than one event listener per element for an event. If you then call the `stopPropagation()` method in one of these event listeners, all other event listeners for this element are still executed.

Let's look at the code in Listing 6.32. In this case, the `handleClickEvent1()` and `handleClickEvent2()` functions are registered as event listeners for the `click` event on all `<div>` elements. As in the previous examples, the `handleClickEvent1()` method toggles the `selected` CSS class and thus colors the background color of the respective element gray at first execution.

The `handleClickEvent2()` method works similarly, but the effect is that the respective element gets a dashed frame. Both methods also prevent the event from being forwarded by calling the `stopPropagation()` method.

If you now click on the `<div>` element with the `C`, the `handleClickEvent1()` function is executed first, then the event object is propagated, despite the call of `stopPropagation()`, and the `handleClickEvent2()` function is executed. In other words, the `stopPropagation()` method doesn't prevent the event from being propagated between event listeners registered for the corresponding event on one and the same element. It only prevents the propagation to those event listeners that have been registered on other elements for the event. The result of the program is shown in Figure 6.16.

```
function handleClickEvent1(ev) {
  const e = ev || window.event;
  this.classList.toggle('selected');
  e.stopPropagation();
}
function handleClickEvent2(ev) {
  const e = ev || window.event;
  this.classList.toggle('selected-border');
  e.stopPropagation();
}
function init() {
  const elements = document.querySelectorAll('.level1, .level2,
.level3');
  for(let i=0; i<elements.length; i++) {
    elements[i].addEventListener('click', handleClickEvent1);
    elements[i].addEventListener('click', handleClickEvent2);
  }
}
```

Listing 6.32: An Example of Two Event Handlers Registered First

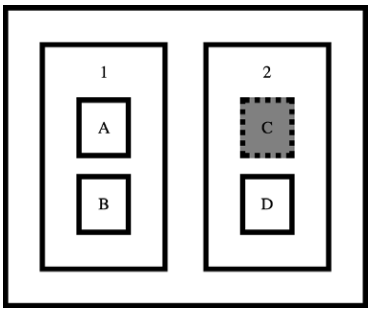


Figure 6.16: Both Registered Event Listeners Called When stopPropagation() Is Called

However, if you want to prevent the propagation of an event even for the event listeners registered on a single element, you must resort to the `stopImmediatePropagation()` method. Listing 6.33 shows an example: the `handleClickEvent1()` function now calls exactly this method, which in turn ensures that the second registered event listener will no longer call the `handleClickEvent2()` function, as shown in Figure 6.17.

```
function handleClickEvent1(ev) {
  const e = ev || window.event;
  this.classList.toggle('selected');
  e.stopImmediatePropagation();
}
function handleClickEvent2(ev) {
  const e = ev || window.event;
```

```
  this.classList.toggle('selected-border');
  e.stopPropagation();
}
```

Listing 6.33: Immediately Canceling the Propagation Also Prevents Other Event Handlers Registered on the Respective Element from Being Executed

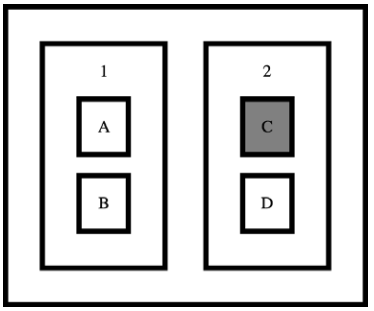


Figure 6.17: By Calling stopImmediatePropagation(), Further Event Listeners on the Same Element Will Not Be Executed

Note

The `stopPropagation()` method prevents events from being propagated to event listeners that are defined for the respective event on other elements, whereas the `stopImmediatePropagation()` method prevents events from being propagated even to those event listeners that were registered for the event on the same element.

6.4.3 Preventing Default Actions of Events

While you can use the `stopPropagation()` and `stopImmediatePropagation()` methods to prevent event propagation, you can use the `preventDefault()` method to prevent the browser's default actions.

An example is shown in Listing 6.34. The `handleLinkClicked()` function is registered as an event listener for the `click` event of a link (i.e., an `<a>` element). However, if the link is clicked, the corresponding linked URL is not opened (which, as we know, would be the browser's default action for handling links); calling the `preventDefault()` method prevents the link from opening.

```
function handleLinkClicked(event) {
  console.log('Link clicked');
  event.preventDefault();
}

function init() {
  const element = document.getElementById('link');
```

```
  element.addEventListener(
    'click',
    handleLinkClicked,
    false
  );
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.34: Preventing Default Actions for Event Listeners

A peculiarity worth pointing out is that, with event handlers (in contrast to event listeners), calling the `preventDefault()` method isn't possible. Instead, this behavior is determined by the return value of the function defined as an event handler: If this value is `false`, the default actions are prevented, but if it's `true` or no return value is specified, the default functions are executed. A corresponding example is shown in Listing 6.35. The `handleLinkClicked()` function, defined as an event handler, returns `false` here, which prevents the browser from opening the URL of the clicked link.

```
function handleLinkClicked(e) {
  console.log('Link clicked');
  return false;
}

function init() {
  const element = document.getElementById('link');
  element.onclick = handleLinkClicked;
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.35: Preventing Default Actions for Event Handlers

Generic Helper Function

A generic helper function that causes both event listeners and event handlers to prevent default browser actions is shown in Listing 6.36. This function first checks whether the `preventDefault()` method exists on the event object. If so, the method is called. Otherwise, the return value of the (now assumed) event handler is set to `false` via the `returnValue` property. The use of this helper function is shown in Listing 6.37.

```
function stopDefault(event) {
  if (event && event.preventDefault) {
    event.preventDefault();
  } else if (window.event && window.event.returnValue) {
    window.event.returnValue = false;
```

```
  }
}
```

Listing 6.36: Helper Function for Preventing Default Actions

```
----------------------------------------

function handleLinkClicked(event) {
  console.log('Link clicked');
  stopDefault(event);
}

function init() {
  const element = document.getElementById('link');
  element.onclick = handleLinkClicked;
  const element2 = document.getElementById('link2');
  element2.addEventListener(
    'click',
    handleLinkClicked,
    false
  );
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.37: Usage of the Helper Function

Return Value of Event Handlers

The return value `false` in a function defined as an event handler prevents not only the default actions of the browser that are normally executed for the respective event, but also the propagation of the event. The return value `false` is virtually equivalent to calling `preventDefault()` and `stopPropagation()` for event listeners.

6.5 Programmatically Triggering Events

Events in JavaScript (and in event-driven programming in general) can be triggered not only by immediate user actions, but also programmatically—that is, by JavaScript code.

6.5.1 Triggering Simple Events

To programmatically trigger an event, simply create an object instance of `Event` and use it as an argument to the `dispatchEvent()` method available to any element in the DOM tree.

The first argument passed to the `Event` constructor is a string that defines the type of the event. Optionally, you can use a configuration object passed as the second argument to determine whether the event should go through the bubbling phase or not (via the `bubbles` property) and whether it is allowed to prevent the browser's default actions (via the `cancelable` property).

The object instance created via the constructor can then be passed to the `dispatchEvent()` method, which internally ensures that this event is triggered on the element on which you call the method. An example is shown in Listing 6.38.

```
function init() {
  const element = document.getElementById('example')
  const event = new Event(
  'example',              // Type of the event
  {
    bubbles: true,        // Allow bubbling
    cancelable: false     // Default actions cannot ...
                          // ... be prevented.
  });
  element.addEventListener(
    'example',                              // Type of event
    (event) => {
      console.log('Event triggered');
      console.log(event.type);              // "example"
    },
    false
  );
  element.dispatchEvent(event);
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.38: Example of Triggering an Event

6.5.2 Triggering Events with Passed Arguments

To assign the triggered event or the corresponding event object-specific information that can then be accessed within the defined event handlers and registered event listeners, use the `CustomEvent` object type instead of the `Event` object type.

The constructor of `CustomEvent` can be passed corresponding data within the configuration object (which can also be passed as a second argument) via the `detail` property. An example is shown in Listing 6.39. In this case, an object (with the properties `firstName`, `lastName`, and `id`) is passed as an argument to the event object and read within the event listener.

```
function init() {
  const element = document.getElementById('example')
  const event = new CustomEvent('example', {
    detail: {
      firstName: 'John',
      lastName: 'Doe',
      id: 4711
    }
  });
  element.addEventListener(
    'example',
    (event) => {
      console.log('Event triggered');
      console.log(event.type);             // "example"
      console.log(event.detail.firstName); // "John"
      console.log(event.detail.lastName);  // "Doe"
      console.log(event.detail.id);        // 4711
    },
    false
  );
  element.dispatchEvent(event);
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.39: Example of Specifying Parameters When Triggering an Event

6.5.3 Triggering Default Events

In addition, you can programmatically trigger other events (e.g., mouse events or keyboard events) and thus virtually simulate user interactions in this way. For this task, simply create an object instance of the respective event object type, as shown in Listing 6.40, using a mouse event, and pass the object instance to the `dispatchEvent()` method.

```
function init() {
  const element = document.getElementById('example')
  const event = new MouseEvent('click', {
    'view': window,
    'bubbles': true,
    'cancelable': true
  });
  element.addEventListener(
    'click',
    (event) => {
      console.log('Element clicked');
```

```
    },
    false
  );
  element.dispatchEvent(event);
}

document.addEventListener('DOMContentLoaded', init);
```

Listing 6.40: Example of Triggering an Event for Mouse Clicks

Browser Support

One final note to conclude this chapter: Support for the events presented in this chapter (and other events not presented here) can vary between browsers. As always, when it comes to browser support of any features, the Can I Use? website at *http://caniuse.com* provides a good overview.

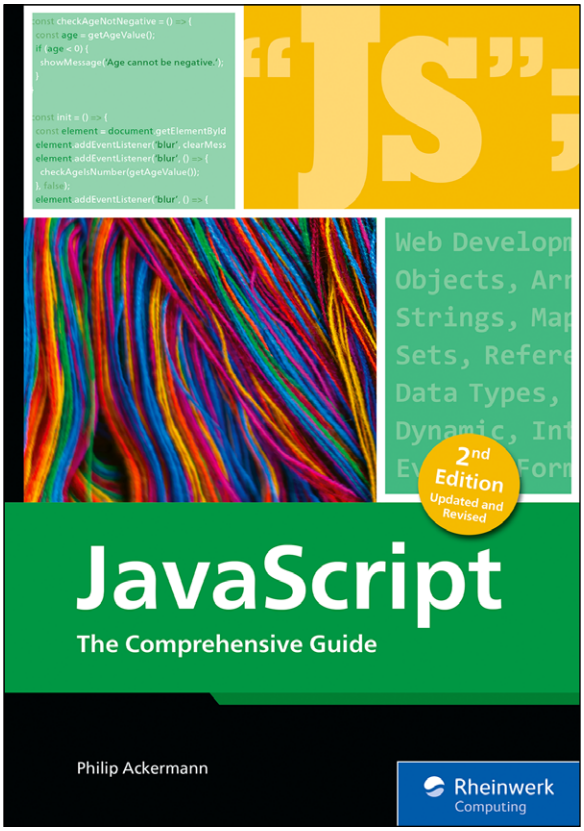
6.6 Summary

In this chapter, you learned about the basic operation of event-driven programming in JavaScript. Among other things, you now know how to respond to user actions, whether mouse or keyboard interactions, form inputs, or other events that are triggered. The following are the most important points to remember from this chapter:

- In *event-driven programming*, events are triggered that can be acted upon within the code.
- In general, the code that responds to events is called an *event handler* or an *event listener*.
- In JavaScript, you can define event handlers using HTML (*HTML event handlers*) or JavaScript (*JavaScript event handlers*), but event listeners can only be defined via JavaScript.
- The advantage of event listeners over event handlers (in JavaScript) is that you can register several for one event.
- In this chapter, we have mainly shown how you can register event handlers/event listeners on elements in the DOM tree. In general, however, the concept of registering event handlers/event listeners goes further; they can also be registered to objects such as the `window` object.
- There are several types of events in JavaScript web development, including mouse events, keyboard events, form interaction events, and many more.
- When an event is triggered on an element in the DOM tree, it goes through different phases: In the *capturing phase*, starting at the document node, the event is passed downwards element by element to the target element, executing each of the event listeners

registered for the capturing phase. In the *target phase*, the event handlers/event listeners registered on the target element are then executed. In the third phase, the *bubbling phase*, the event rises back up to the document node starting from the target element, executing all event handlers/event listeners registered for this phase.

- Last but not least, you learned how to trigger events programmatically, whether via the `Event` object type to trigger general events, via the `CustomEvent` object type to trigger events using arguments, or via the various other object types such as `MouseEvent`.



Philip Ackermann

JavaScript

The Comprehensive Guide

- Work with objects, reference types, events, forms, and web APIs
- Build server-side applications, mobile applications, AI applications, and more
- Download and consult practical code examples

www.sap-press.com/6242

The Author

Philip Ackermann is the CTO of Cedalo GmbH and the author of several reference books and technical articles on Java, JavaScript, and web development. His focus is on the design and development of Node.js projects in the areas of Industry 4.0 and the Internet of Things.

ISBN 978-1-4932-2797-6 • 973 pages • 09/2026

E-book: $64.99 • **Print book:** $69.95 • **Bundle:** $79.99

Rheinwerk Publishing