Events and Callbacks

In the browser most code is **event–driven** and writing interactive applications in JavaScript is often about waiting for and reacting to events, to alter the behavior of the browser in some way. Events occur when the page loads, when user interacts (clicks, hovers, changes) and myriad other times, and can be triggered manually too.
To react to an event you **listen** for it and supply a function which will be called by the browser when the event occurs. This function is known as a **callback**.

See the example below:

```
var handleClick = function (event) {
    // do something!
};
var button = document.querySelector('#big-button');
button.addEventListener('click', handleClick);
```

addEventListener is a method found on all DOM elements. Here it's being called on an element saved in the button variable. The first argument is a string – the name of the **event** to listen for. Here's it's click, that's a click of the mouse or a tap of the finger. The second is the **callback** function – here it's handleClick.

It is important to note that Internet Explorer does not support **addEventListener** in versions earlier than 9. Instead attachEvent must be used as demostrated below:

```
button.attachEvent('onclick', handleClick);
```

Data about a particular event is passed to the event callback. Take a look at handleClick, declared above. You can see its argument: event – it's an object whose properties describe what occurred.

Here's an example event you might see into a click event callback like handleClick. There are lots of properties giving you an idea of where the event occurred on the page (like pageX and offsetY) – these are slightly different because they depend on the reference point used to measure from. You can also see the target which is a reference to the node that was clicked.


```
{
    offsetX: 74,
    offsetY: 10,
    pageX: 154,
    pageY: 576,
    screenX: 154,
    screenY: 489,
    target: h2,
    timeStamp: 1363131952985,
    type: "click",
    x: 154,
    y: 395
}
```

Pointer events

There are currently two widely used ways to point at things on a screen: mice (including devices that act like mice, such as touchpads and trackballs) and touchscreens. These produce different kinds of events.

–Mouse clicks

Pressing a mouse button causes a number of events to fire. The "mousedown" and "mouseup" events are similar to "keydown" and "keyup" and fire when the button is pressed and released. These happen on the DOM nodes that are immediately below the mouse pointer when the event occurs.

After the "mouseup" event, a "click" event fires on the most specific node that contained both the press and the release of the button. For example, if I press down the mouse button on one paragraph and then move the pointer to another paragraph and release the button, the "click" event will happen on the element that contains both those paragraphs.

If two clicks happen close together, a "dblclick" (double–click) event also fires, after the second click event.

To get precise information about the place where a mouse event happened, you can look at its clientX and clientY properties, which contain the event's coordinates (in pixels) relative to the top–left corner of the window, or pageX and pageY, which are relative to the top–left corner of the whole document (which may be different when the window has been scrolled).

The following implements a primitive drawing program. Every time you click the document, it adds a dot under your mouse pointer. See Chapter 19 for a less primitive drawing program.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

– **Mouse motion**

Every time the mouse pointer moves, a "mousemove" event is fired. This event can be used to track the position of the mouse. A common situation in which this is useful is when implementing some form of mouse–dragging functionality.

As an example, the following program displays a bar and sets up event handlers so that dragging to the left or right on this bar makes it narrower or wider:

```
<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // Tracks the last observed mouse X position
  let bar = document.querySelector("div");
  bar.addEventListener("mousedown", event => {
    if (event.button == 0) {
      lastX = event.clientX;
      window.addEventListener("mousemove", moved);
      event.preventDefault(); // Prevent selection
    }
  });

  function moved(event) {
    if (event.buttons == 0) {
```

```
      window.removeEventListener("mousemove", moved);
    } else {
      let dist = event.clientX - lastX;
      let newWidth = Math.max(10, bar.offsetWidth + dist);
      bar.style.width = newWidth + "px";
      lastX = event.clientX;
    }
  }
</script>
```

Note that the "mousemove" handler is registered on the whole window. Even if the mouse goes outside of the bar during resizing, as long as the button is held we still want to update its size.

We must stop resizing the bar when the mouse button is released. For that, we can use the buttons property (note the plural), which tells us about the buttons that are currently held down. When this is zero, no buttons are down. When buttons are held, its value is the sum of the codes for those buttons—the left button has code 1, the right button 2, and the middle one 4. That way, you can check whether a given button is pressed by taking the remainder of the value of buttons and its code.

Note that the order of these codes is different from the one used by button, where the middle button came before the right one. As mentioned, consistency isn't really a strong point of the browser's programming interface.

– Touch events

The style of graphical browser that we use was designed with mouse interfaces in mind, at a time where touchscreens were rare. To make the Web “work” on early touchscreen phones, browsers for those devices pretended, to a certain extent, that touch events were mouse events. If you tap your screen, you’ll get "mousedown", "mouseup", and "click" events.

But this illusion isn’t very robust. A touchscreen works differently from a mouse: it doesn’t have multiple buttons, you can’t track the finger when it isn’t on the screen (to simulate "mousemove"), and it allows multiple fingers to be on the screen at the same time.

Mouse events cover touch interaction only in straightforward cases—if you add a "click" handler to a button, touch users will still be able to use it. But something like the resizeable bar in the previous example does not work on a touchscreen.

There are specific event types fired by touch interaction. When a finger starts touching the screen, you get a "touchstart" event. When it is moved while touching, "touchmove" events fire. Finally, when it stops touching the screen, you’ll see a "touchend" event.

Because many touchscreens can detect multiple fingers at the same time, these events don’t have a single set of coordinates associated with them. Rather, their event objects have a touches property, which holds an array-like object of points, each of which has its own clientX, clientY, pageX, and pageY properties.

You could do something like this to show red circles around every touching finger:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border–radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Touch this page</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX – 50) + "px";
      dot.style.top = (pageY – 50) + "px";
      document.body.appendChild(dot);
    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>
```

You'll often want to call preventDefault in touch event handlers to override the browser's default behavior (which may include scrolling the page on swiping) and to prevent the mouse events from being fired, for which you may also have a handler.

– Scroll events

Whenever an element is scrolled, a "scroll" event is fired on it. This has various uses, such as knowing what the user is currently looking at (for disabling off-screen animations or sending spy reports to your evil headquarters) or showing some indication of progress (by highlighting part of a table of contents or showing a page number).

The following example draws a progress bar above the document and updates it to fill up as you scroll down:

```
<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Create some content
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));
```

```
  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Giving an element a position of fixed acts much like an absolute position but also prevents it from scrolling along with the rest of the document. The effect is to make our progress bar stay at the top. Its width is changed to indicate the current progress. We use %, rather than px, as a unit when setting the width so that the element is sized relative to the page width.

The global innerHeight binding gives us the height of the window, which we have to subtract from the total scrollable height—you can't keep scrolling when you hit the bottom of the document. There's also an innerWidth for the window width. By dividing pageYOffset, the current scroll position, by the maximum scroll position and multiplying by 100, we get the percentage for the progress bar.

Calling preventDefault on a scroll event does not prevent the scrolling from happening. In fact, the event handler is called only after the scrolling takes place.

– **Focus events**

When an element gains focus, the browser fires a "focus" event on it. When it loses focus, the element gets a "blur" event.

Unlike the events discussed earlier, these two events do not propagate. A handler on a parent element is not notified when a child element gains or loses focus.

The following example displays help text for the text field that currently has focus:

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

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

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


<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    }); }
</script>
```

The window object will receive "focus" and "blur" events when the user moves from or to the browser tab or window in which the document is shown.

– **Load event**.

When a page finishes loading, the "load" event fires on the window and the document body objects. This is often used to schedule initialization actions that require the whole document to have been built. Remember that the content of <script> tags is run immediately when the tag is encountered. This may be too soon, for example when the script needs to do something with parts of the document that appear after the <script> tag.

Elements such as images and script tags that load an external file also have a "load" event that indicates the files they reference were loaded. Like the focus–related events, loading events do not propagate.

When a page is closed or navigated away from (for example, by following a link), a "beforeunload" event fires. The main use of this event is to prevent the user from accidentally losing work by closing a document. If you prevent the default behavior on this event and set the returnValue property on the event object to a string, the browser will show the user a dialog asking if they really want to leave the page. That dialog might include your string, but because some malicious sites try to use these dialogs to confuse people into staying on their page to look at dodgy weight loss ads, most browsers no longer display them.

– **Events and the event loop**

In the context of the event loop, browser event handlers behave like other asynchronous notifications. They are scheduled when the event occurs but must wait for other scripts that are running to finish before they get a chance to run.

The fact that events can be processed only when nothing else is running means that, if the event loop is tied up with other work, any interaction with the page (which happens through events) will be delayed until there's time to process it. So if you schedule too much work, either with long-running event handlers or with lots of short-running ones, the page will become slow and cumbersome to use.

For cases where you really do want to do some time-consuming thing in the background without freezing the page, browsers provide something called web workers. A worker is a JavaScript process that runs alongside the main script, on its own timeline.

Imagine that squaring a number is a heavy, long-running computation that we want to perform in a separate thread. We could write a file called code/squareworker.js that responds to messages by computing a square and sending a message back.

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

To avoid the problems of having multiple threads touching the same data, workers do not share their global scope or any other data with the main script's environment. Instead, you have to communicate with them by sending messages back and forth.

This code spawns a worker running that script, sends it a few messages, and outputs the responses.

```
let squareWorker = new Worker("code/squareworker.js");

squareWorker.addEventListener("message", event => {

  console.log("The worker responded:", event.data);

});

squareWorker.postMessage(10);

squareWorker.postMessage(24);
```

The postMessage function sends a message, which will cause a "message" event to fire in the receiver. The script that created the worker sends and receives messages through the Worker object, whereas the worker talks to the script that created it by sending and listening directly on its global scope. Only values that can be represented as JSON can be sent as messages—the other side will receive a copy of them, rather than the value itself.

Timers

Timers schedules another function to be called later, after a given number of milliseconds.

Sometimes you need to cancel a function you have scheduled. This is done by storing the value returned by setTimeout and calling clearTimeout on it.

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

The cancelAnimationFrame function works in the same way as clearTimeout—calling it on a value returned by requestAnimationFrame will cancel that frame (assuming it hasn't already been called).

A similar set of functions, setInterval and clearInterval, are used to set timers that should repeat every X milliseconds.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

– **Debouncing**

Some types of events have the potential to fire rapidly, many times in a row (the "mousemove" and "scroll" events, for example). When handling such events, you must be careful not to do anything too time-consuming or your handler will take up so much time that interaction with the document starts to feel slow.

If you do need to do something nontrivial in such a handler, you can use setTimeout to make sure you are not doing it too often. This is usually called debouncing the event. There are several slightly different approaches to this.

In the first example, we want to react when the user has typed something, but we don't want to do it immediately for every input event. When they are typing quickly, we just want to wait until a pause occurs. Instead of immediately performing an action in the event handler, we set a timeout. We also clear the previous timeout (if any) so that when events occur close together (closer than our timeout delay), the timeout from the previous event will be canceled.

```
<textarea>Type something here...</textarea>

<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);});
</script>
```

Giving an undefined value to clearTimeout or calling it on a timeout that has already fired has no effect. Thus, we don't have to be careful about when to call it, and we simply do so for every event.

We can use a slightly different pattern if we want to space responses so that they're separated by at least a certain length of time but want to fire them during a series of events, not just afterward. For example, we might want to respond to "mousemove" events by showing the current coordinates of the mouse but only every 250 milliseconds.

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```

Summary

Event handlers make it possible to detect and react to events happening in our web page. The addEventListener method is used to register such a handler.

Each event has a type ("keydown", "focus", and so on) that identifies it. Most events are called on a specific DOM element and then propagate to that element's ancestors, allowing handlers associated with those elements to handle them.

When an event handler is called, it is passed an event object with additional information about the event. This object also has methods that allow us to stop further propagation (stopPropagation) and prevent the browser's default handling of the event (preventDefault).

Pressing a key fires "keydown" and "keyup" events. Pressing a mouse button fires "mousedown", "mouseup", and "click" events. Moving the mouse fires "mousemove" events. Touchscreen interaction will result in "touchstart", "touchmove", and "touchend" events.

Scrolling can be detected with the "scroll" event, and focus changes can be detected with the "focus" and "blur" events. When the document finishes loading, a "load" event fires on the window.

This is not all about event–driven programming we will cover more I in a later day.

Best Wishes.

Lux Academy