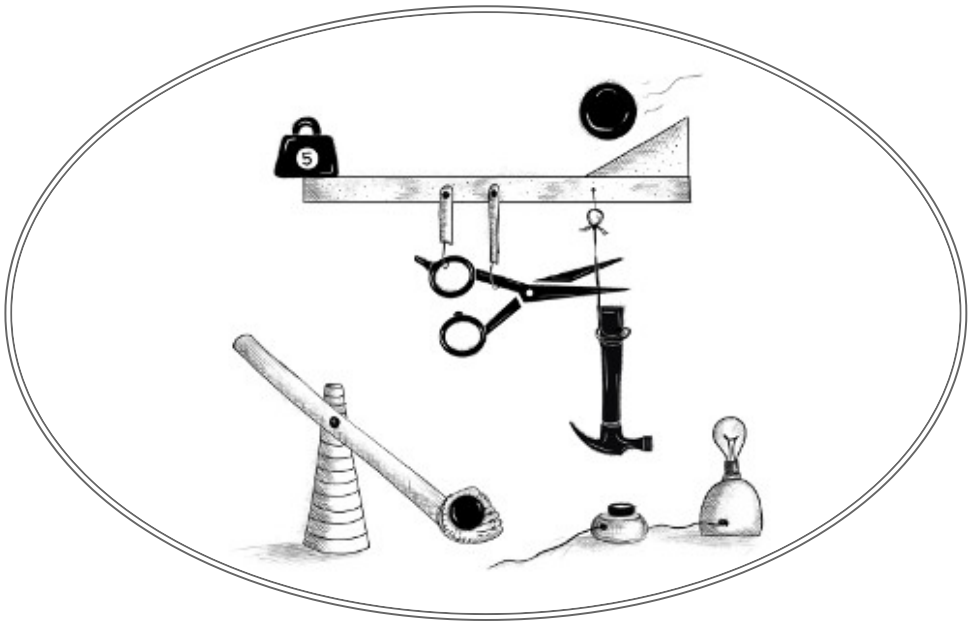
Handling Events

> "You have power over your mind—not outside events. Realize this, and you will find strength."
>
> —Marcus Aurelius, *Meditations*



Some programs work with direct user input, such as mouse and keyboard actions. That kind of input isn't available ahead of time, as a well-organized data structure—it comes in piece by piece, in real time, and the program must respond to it as it happens.

Event handlers

Imagine an interface where the only way to find out whether a key on the keyboard is being pressed was to read the current state of that key. To be able to react to keypresses, you would have to constantly read the key's state to catch it before it was released again. It would be dangerous to perform other time-intensive computations, since you might miss a keypress.

Some primitive machines handle input like this. A step up from this is for the hardware or operating system to notice the keypress and put it in a queue. A

program can then periodically check the queue for new events and react to what it finds there.

Of course, the program has to remember to look at the queue, and to do it often, because any time between the key being pressed and the program noticing the event will cause the software to feel unresponsive. This approach is called *polling*. Most programmers prefer to avoid it.

A better mechanism is for the system to actively notify the code when an event occurs. Browsers do this by allowing us to register functions as *handlers* for specific events.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

The `window` binding refers to a built-in object provided by the browser. It represents the browser window that contains the document. Calling its `addEventListener` method registers the second argument to be called whenever the event described by its first argument occurs.

Events and DOM nodes

Each browser event handler is registered in a context. In the previous example, we called `addEventListener` on the `window` object to register a handler for the whole window. Such a method can also be found on DOM elements and some other types of objects. Event listeners are called only when the event happens in the context of the object on which they are registered.

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

That example attaches a handler to the button node. Clicks on the button cause that handler to run, but clicks on the rest of the document do not.

Giving a node an `onclick` attribute has a similar effect. This works for most types of events—you can attach a handler through the attribute whose name is the event name with `on` in front of it.

But a node can have only one `onclick` attribute, so you can register only one handler per node that way. The `addEventListener` method allows you to add any number of handlers meaning it's safe to add handlers even if there is already another handler on the element.

The `removeEventListener` method, called with arguments similar to `addEventListener`, removes a handler.

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

The function given to `removeEventListener` has to be the same function value given to `addEventListener`. When you need to unregister a handler, you'll want to give the handler function a name (`once`, in the example) to be able to pass the same function value to both methods.

Event objects

Though we have ignored it so far, event handler functions are passed an argument: the *event object*. This object holds additional information about the event. For example, if we want to know *which* mouse button was pressed, we can look at the event object's `button` property.

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button");
```

```
    button.addEventListener("mousedown", event => {
      if (event.button == 0) {
        console.log("Left button");
      } else if (event.button == 1) {
        console.log("Middle button");
      } else if (event.button == 2) {
        console.log("Right button");
      }
    });
  </script>
```

The information stored in an event object differs per type of event. (We'll discuss different types later in the chapter.) The object's `type` property always holds a string identifying the event (such as `"click"` or `"mousedown"`).

Propagation

For most event types, handlers registered on nodes with children will also receive events that happen in the children. If a button inside a paragraph is clicked, event handlers on the paragraph will also see the click event.

But if both the paragraph and the button have a handler, the more specific handler—the one on the button—gets to go first. The event is said to *propagate* outward from the node where it happened to that node's parent node and on to the root of the document. Finally, after all handlers registered on a specific node have had their turn, handlers registered on the whole window get a chance to respond to the event.

At any point, an event handler can call the `stopPropagation` method on the event object to prevent handlers further up from receiving the event. This can be useful when, for example, you have a button inside another clickable element and you don't want clicks on the button to activate the outer element's click behavior.

The following example registers `"mousedown"` handlers on both a button and the paragraph around it. When clicked with the right mouse button, the handler for the button calls `stopPropagation`, which will prevent the

handler on the paragraph from running. When the button is clicked with another mouse button, both handlers will run.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

Most event objects have a `target` property that refers to the node where they originated. You can use this property to ensure that you're not accidentally handling something that propagated up from a node you do not want to handle.

It is also possible to use the `target` property to cast a wide net for a specific type of event. For example, if you have a node containing a long list of buttons, it may be more convenient to register a single click handler on the outer node and have it use the `target` property to figure out whether a button was clicked, rather than registering individual handlers on all of the buttons.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent);
    }
  });
</script>
```

Default actions

Many events have a default action. If you click a link, you will be taken to the link's target. If you press the down arrow, the browser will scroll the page down. If you right-click, you'll get a context menu. And so on.

For most types of events, the JavaScript event handlers are called *before* the default behavior takes place. If the handler doesn't want this normal behavior to happen, typically because it has already taken care of handling the event, it can call the `preventDefault` method on the event object.

This can be used to implement your own keyboard shortcuts or context menu. It can also be used to obnoxiously interfere with the behavior that users expect. For example, here is a link that cannot be followed:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

Try not to do such things without a really good reason. It'll be unpleasant for people who use your page when expected behavior is broken.

Depending on the browser, some events can't be intercepted at all. On Chrome, for example, the keyboard shortcut to close the current tab (CTRL-W or COMMAND-W) cannot be handled by JavaScript.

Key events

When a key on the keyboard is pressed, your browser fires a `"keydown"` event. When it is released, you get a `"keyup"` event.

```
<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
```

```
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

Despite its name, `"keydown"` fires not only when the key is physically pushed down. When a key is pressed and held, the event fires again every time the key *repeats*. Sometimes you have to be careful about this. For example, if you add a button to the DOM when a key is pressed and remove it again when the key is released, you might accidentally add hundreds of buttons when the key is held down longer.

The previous example looks at the `key` property of the event object to see which key the event is about. This property holds a string that, for most keys, corresponds to the thing that pressing that key would type. For special keys such as ENTER, it holds a string that names the key (`"Enter"`, in this case). If you hold SHIFT while pressing a key, that might also influence the name of the key—`"v"` becomes `"V"`, and `"1"` may become `"!"`, if that is what pressing SHIFT-1 produces on your keyboard.

Modifier keys such as SHIFT, CTRL, ALT, and META (COMMAND on Mac) generate key events just like normal keys. When looking for key combinations, you can also find out whether these keys are held down by looking at the `shiftKey`, `ctrlKey`, `altKey`, and `metaKey` properties of keyboard and mouse events.

```
<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
    }
  });
</script>
```

The DOM node where a key event originates depends on the element that has focus when the key is pressed. Most nodes cannot have focus unless you give them a `tabindex` attribute, but things like links, buttons, and form fields can.

We'll come back to form fields in Chapter 18. When nothing in particular has focus, `document.body` acts as the target node of key events.

When the user is typing text, using key events to figure out what is being typed is problematic. Some platforms, most notably the virtual keyboard on Android phones, don't fire key events. But even when you have an old-fashioned keyboard, some types of text input don't match key presses in a straightforward way, such as *input method editor* (*IME*) software used by people whose scripts don't fit on a keyboard, where multiple key strokes are combined to create characters.

To notice when something was typed, elements that you can type into, such as the `<input>` and `<textarea>` tags, fire `"input"` events whenever the user changes their content. To get the actual content that was typed, it is best to directly read it from the focused field, which we discuss in Chapter 18.

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 program implements a primitive drawing application. Every time you click the document, it adds a dot under your mouse pointer.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: teal;
    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>
```

We'll create a less primitive drawing application in Chapter 19.

Mouse motion

Every time the mouse pointer moves, a `"mousemove"` event fires. 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 it is zero, no buttons are down. When buttons are held, the value of the `buttons` property is the sum of the codes for those buttons—the left button has code 1, the right button 2, and the middle one 4. With the left and right buttons held, for example, the value of `buttons` will be 3.

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 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 doesn't work like 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 must 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 you close page or navigate away from it (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, as discussed in Chapter 11, 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

The `setTimeout` function we saw in Chapter 11 schedules another function to be called later, after a given number of milliseconds. Sometimes you need to cancel a function you have scheduled. You can do this 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, such as the `"mousemove"` and `"scroll"` events. 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.

For example, suppose 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's 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.

Exercises

Balloon

Write a page that displays a balloon (using the balloon emoji, 🎈). When you press the up arrow, it should inflate (grow) 10 percent. When you press the down arrow, it should deflate (shrink) 10 percent.

You can control the size of text (emoji are text) by setting the `font-size` CSS property (`style.fontSize`) on its parent element. Remember to include a unit in the value—for example, pixels (`10px`).

The key names of the arrow keys are `"ArrowUp"` and `"ArrowDown"`. Make sure the keys change only the balloon, without scrolling the page.

Once you have that working, add a feature where if you blow up the balloon past a certain size, it "explodes". In this case, exploding means that it is replaced with an 💥 emoji, and the event handler is removed (so that you can't inflate or deflate the explosion).

```
<p>🎈</p>

<script>
  // Your code here
</script>
```

▶ Display hints...

Mouse trail

In JavaScript's early days, which was the high time of gaudy home pages with lots of animated images, people came up with some truly inspiring ways to use the language. One of these was the *mouse trail*—a series of elements that would follow the mouse pointer as you moved it across the page.

In this exercise, I want you to implement a mouse trail. Use absolutely positioned `<div>` elements with a fixed size and background color (refer to the code in the "Mouse Clicks" section for an example). Create a bunch of these elements and, when the mouse moves, display them in the wake of the mouse pointer.

There are various possible approaches here. You can make your trail as simple or as complex as you want. A simple solution to start with is to keep a fixed number of trail elements and cycle through them, moving the next one to the mouse's current position every time a `"mousemove"` event occurs.

```
<style>
  .trail { /* className for the trail elements */
    position: absolute;
    height: 6px; width: 6px;
    border-radius: 3px;
    background: teal;
  }
  body {
    height: 300px;
  }
</style>

<script>
  // Your code here.
</script>
```

▶ Display hints...

Tabs

Tabbed panels are common in user interfaces. They allow you to select an interface panel by choosing from a number of tabs “sticking out” above an element.

Implement a simple tabbed interface. Write a function, `asTabs`, that takes a DOM node and creates a tabbed interface showing the child elements of that node. It should insert a list of `<button>` elements at the top of the node, one for each child element, containing text retrieved from the `data-tabname` attribute of the child. All but one of the original children should be hidden (given a `display` style of `none`). The currently visible node can be selected by clicking the buttons.

When that works, extend it to style the button for the currently selected tab differently so that it is obvious which tab is selected.

```
<tab-panel>
  <div data-tabname="one">Tab one</div>
  <div data-tabname="two">Tab two</div>
  <div data-tabname="three">Tab three</div>
</tab-panel>
<script>
  function asTabs(node) {
    // Your code here.
  }
  asTabs(document.querySelector("tab-panel"));
</script>
```

▶ Display hints...

◂ ● ▸ ?