Event

Bubbling and Delegation

Learning Goals

- Understand event bubbling and delegation
- Learn to use event bubbling to set event listeners

Event Basics

Events are happening all the time in the browser. When the browser has finished loading the page, an event is fired.

Every time the user moves their mouse, hovers over an element, clicks or taps, submits a form, presses down on a key or takes their finger off that key — an event is fired.

Some of these events are very easy to spot when they occur (e.g. the user clicks on a hyperlink), but many go by completely unnoticed.

It is, however, possible for us to use JavaScript to set up listeners for events that interest us. Our listeners wait patiently on a DOM node until the event they're waiting for is fired.

Then, they spring into action, running an appropriate function to respond to the event as required, or whatever else you deem appropriate.

For a review of how to set event listeners, please refer to the Introduction to JavaScript III - Intro to DOM Manipulation

Event Bubbling

Now we've talked about the very basics of events, let's turn our attention to event bubbling, which refers to the ability of events set on DOM nodes to "bubble up" and also apply to children of those nodes. We'll start with a quick experiment.

Experiment

See CodePen in markdown.

Pair Practice

Pair Practice

- Add a click event to the button, that logs the element that was clicked on using `this`.

Pair Practice

- Add a click event to the button, that logs the element that was clicked on using `this`.
- Move the event listener to the `.parent` element. What is the result when you click on the button?

Pair Practice

- Add a click event to the button, that logs the element that was clicked on using `this`.
- Move the event listener to the `.parent` element. What is the result when you click on the button?
- Move the event listener from the first step to the `.grandparent` element.
  - What is the result when you click on the button?
  - What is is the result when you click the `.parent` element?

Discussion

You may have noticed that the event listeners on a parent element are fired whenever the action occurs on one of its children.

When an event occurs:

When an event occurs:

— The browser checks the element to see if there are any event listeners registered.

When an event occurs:

- The browser checks the element to see if there are any event listeners registered.
- After it checks the element where the event occurred, the browser works its way up the DOM tree to see if any of the parents have a listener registered
  - then grandparents, and so on.

When an event occurs:

- The browser checks the element to see if there are any event listeners registered.
- After it checks the element where the event occurred, the browser works its way up the DOM tree to see if any of the parents have a listener registered
  - then grandparents, and so on.
- It checks every element all the way up to the root (see `event.path`).

When an event occurs:

- The browser checks the element to see if there are any event listeners registered.
- After it checks the element where the event occurred, the browser works its way up the DOM tree to see if any of the parents have a listener registered
  - then grandparents, and so on.
- It checks every element all the way up to the root (see `event.path`).
- This process is known as:

Event Bubbling

Try out the following code in your forked CodePen:

```
document.querySelector('.grandparent').addEventListener('click', function (event) {
  console.log('Grandparent');
});

document.querySelector('.parent').addEventListener('click', function (event) {
  console.log('Parent');
});

document.querySelector('#click-me').addEventListener('click', function (event) {
  console.log('Button');
});
```

If you click on the button, you'll see that the events all bubble up through the `.parent` and `.grandparent` elements — this provides a more explicit proof than the solutions you may come up with for the previous question.

The Event Object

The anonymous function passed to `document.addEventListener()` takes an optional argument, which it assigns an `Event` object to. In the case of the click event we've been using as an example, this is a `MouseEvent`. You can visit the MDN page for `Event` to explore the full list of supported event types.

Each type of event supports a number of different properties.

`MouseEvent` has information about the `x` and `y` coordinates where the mouse was clicked.

`KeyboardEvent` has information about which key was pressed.

The `target` and `currentTarget` properties on the `Event` object can be useful during the event bubbling phase.

The `target` property represents the node that actually triggered the event, whereas `currentTarget` is the node that is listening for the event. Sometimes, `target` and `currentTarget` are the same node, but not always.

For example, if a `div` listens for a `click` event, a child button of the `div` is clicked, the `target` would be the button, and the `currentTarget` would be the `div`.

```html
<!-- index.html -->
<div class="container">
  <button class="submit">Submit</button>
</div>
```

```js
// scripts.js
document.querySelector(".container")
  .addEventListener("click", function (event) {
    console.log(event.currentTarget) // With only one event listener registered,
      // this will always be the div.container node

    console.log(event.target) // if you click the button, this will be the
      // button.submit node
      // if you click outside of the button, but still in the .container space,
      // it will log the div.container node
  })
```

Let's make some changes to the code from earlier. Instead of logging a description of each element where an event was triggered, either by a click or through event bubbling, let's log the `target` and `currentTarget` of the event.

```
document.querySelector('.grandparent').addEventListener('click', function (event) {
  console.log("target", event.target);
  console.log("currentTarget", event.currentTarget);
});

document.querySelector('.parent').addEventListener('click', function (event) {
  console.log("target", event.target);
  console.log("currentTarget", event.currentTarget);
});

document.querySelector('#click-me').addEventListener('click', function (event) {
  console.log("target", event.target);
  console.log("currentTarget", event.currentTarget);
});
```

STOP && REFACTOR

Just because we are in JavaScriptopolis doesn't mean we have to write WET ("Woo! Extra Typing!") code! In the interest of always keeping an eye out for repetitive tasks, let's see that previous snippet refactored with a `forEach` call:

```
const selectors = [".grandparent", ".parent", "#click-me"]

selectors.forEach(function (selector) {
  document.querySelector(selector)
    .addEventListener("click", function (event) {
      console.log("target", event.target)
      console.log("currentTarget", event.currentTarget)
    })
})
```

Let's demonstrate how nice and easily separated JavaScript callback functions are:

```
const selectors = [".grandparent", ".parent", "#click-me"]

selectors.forEach(listenForClick)

function listenForClick (selector) {
  const targets = ["target", "currentTarget"]
  document.querySelector(selector)
    .addEventListener("click", logger(targets))
}

function logger (targets) {
  return function (event) {
    targets.forEach(target => console.log(target, event[target]))
  }
}
```

Yes, more lines of code. **BUT**

Yes, more lines of code. **BUT**

— **Wayyy** easier to test.

Yes, more lines of code. **BUT**

— **Wayyy** easier to test.
— **Wayyy** more flexible over time.

Yes, more lines of code. **BUT**

— **Wayyy** easier to test.

— **Wayyy** more flexible over time.

— **Wayyy** more reusable.

Yes, more lines of code. **BUT**

- **Wayyy** easier to test.
- **Wayyy** more flexible over time.
- **Wayyy** more reusable.
- More about telling your code what to do rather than how to do it (i.e., declarative vs imperative programming).

Back
to
Events

Pair Practice

Modify the previous code to log the `event` itself (as opposed to the `target` property on the event). What other properties on the event object look particularly useful?

Adding and Removing Event Listeners

A common obstacle that many JavaScript developers struggle with is understanding the timing in which they bind event listeners to DOM nodes. When we add event listeners to DOM nodes, we're only adding them to the nodes that are currently on the page. We are not adding listeners to nodes that may be added to the page in the future.

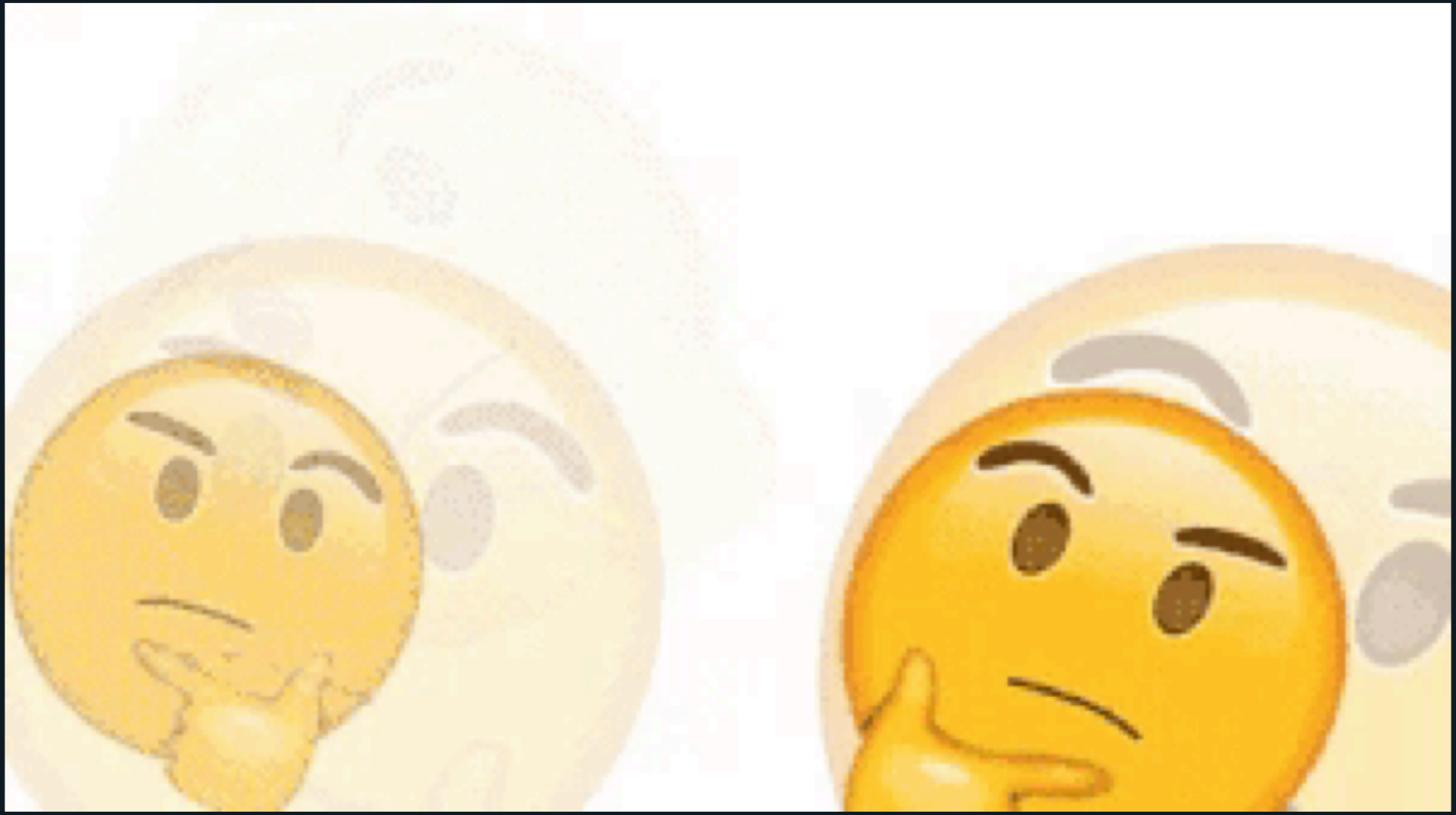
Experiment

More CodePen!

You should see three buttons labeled "Click me!" as well as a button for adding new buttons to the page.

1. Click each of the "Click me!" buttons and verify that each one fires an `alert` notifying you that the button has in fact been clicked.
2. Add an additional button using the "Add a new button below." button.
3. Click on your new button and observe the results.

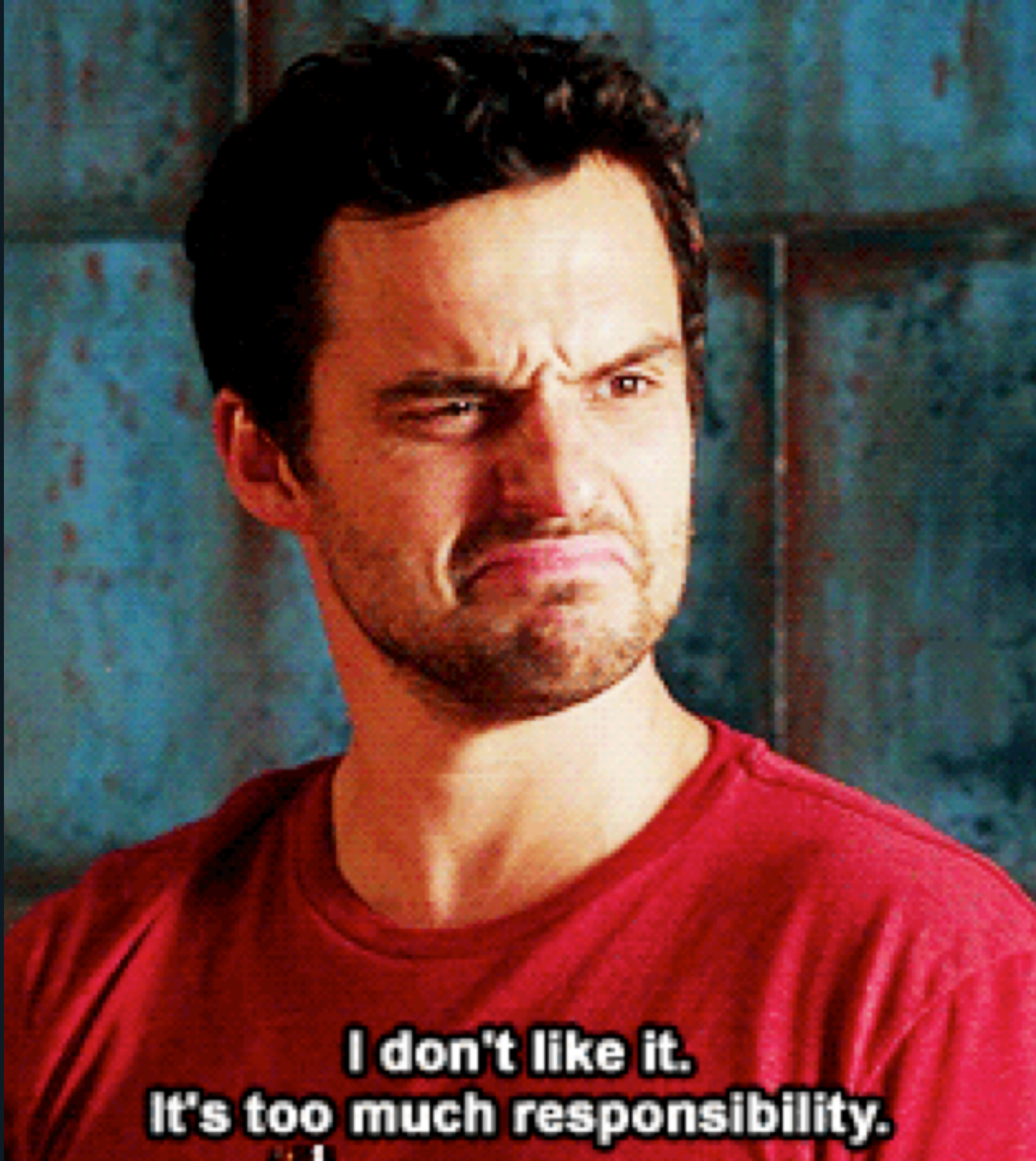
What did you notice?



The event listeners are only bound to the buttons that were present when the page code was first loaded. The buttons we added later were not around when we added the listeners.

What could we do to fix this?

Event Delegation



Setting event listeners on specific newly created DOM nodes is one way to set event listeners. However, if you're not careful, you may end up setting multiple listeners on the same node.

Also, You can cause a memory leak if an event listeners are not unbound from an element when it is removed from the DOM.

Rather than manage the addition and removal of event listeners, there is a methodology you can use called **event delegation**.

In **event delegation**, we take advantage of the fact that events bubble in the event loops by setting an event listener on one parent. This event listener analyzes bubbled events to find a match in its child elements.

Event delegation in tandem with `target` and `currentTarget` allows you to have more articulate control over what events actually execute your JavaScript.

Enough with the Bubbles

You have already come across `event.preventDefault()`, which does exactly what its name implies.

In the context of event bubbling and delegation, there is another function on the `Event` object that is useful for controlling event flow:

```
event.stopPropagation()
```

This prevents the event from traveling up the ancestral node path.

This is helpful if a parent and child both have listeners, but should be executed in different contexts.

I never thought I would ever say the words

"ancestral node path".

Coming back to this generic logging example, what happens if you stop propagation after the `console.log()` line?

```
const selectors = [".grandparent", ".parent", "#click-me"]

selectors.forEach(listenForClick)

function listenForClick (selector) {
  const targets = ["target", "currentTarget"]
  document.querySelector(selector)
    .addEventListener("click", logger(targets))
}

function logger (targets) {
  return function (event) {
    targets.forEach(target => console.log(target, event[target]))
    event.stopPropagation()
  }
}
```

Click around the DOM to see the results.

Pair Practice

Fork the previous CodePen and write a click event handler that responds to (and only to) buttons that both do and could exist on the DOM.

Review

Review

- When an element is bound to an event listener, what besides that element can trigger the event?

Review

- When an element is bound to an event listener, what besides that element can trigger the event?
- How does this relate to "event bubbling"?

Review

- When an element is bound to an event listener, what besides that element can trigger the event?
- How does this relate to "event bubbling"?
- What special object do we have access to when dealing with UI interactions?
  - What are three useful properties on that object?

Review

- When an element is bound to an event listener, what besides that element can trigger the event?
- How does this relate to "event bubbling"?
- What special object do we have access to when dealing with UI interactions?
  - What are three useful properties on that object?
- What is the difference between `target` and `currentTarget`?

Back to QS

Let's check out the first iteration of Quantified Self and see where event delegation would be helpful!

— jQuery has an easy way to do event delegation with the `on` function. Check out your QS project to make sure you're using event delegation. If you aren't, do some refactoring!