

Event Handling: The Bubbling Principle

Bubbles are little pockets of air that make for an amusing time in the bath. Sometimes, though, bubbles can be annoying -- like when they suddenly pop, or when there are too many and they're overflowing! We can think about Javascript events and their handlers as bubbles that rise up through the murky waters of the DOM until they reach the surface, or the top-level DOM element.

It's important for developers to understand The Bubbling Principle and use it to properly handle events and/or to stop events from bubbling up to outer elements and causing unintended effects.

What is the bubbling principle?

According to [this handy bubbling explainer](#) on Javascript.info, The Bubbling Principle means that *when an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors*. Consider the following example HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      window.addEventListener("DOMContentLoaded", event => {
        document.body.addEventListener("click", event => {
          console.log(event.target.id);
        });
      });
    </script>
  </head>
  <body>
    <div onclick="console.log('The onclick handler!')">
      <p id="paragraph">
```

```
      If you click on this P, the onclick event for the DIV actually runs.
    </p>
  </div>
</body>
</html>
```

In the `<script>`, you can see the event listener for `DOMContentLoaded`, and inside it, another listener for a `click` event on the `<body>` element of the document accessed through the special property `document.body`. (You could also use `document.querySelector('body')`, too.) By now, we should be used to listening for click events in our scripts. However, there's another way to run a function on a `click` event as an attribute of the `div` in the body of the HTML, a way that **you should never ever use in real production code!**

Check out that `onclick` attribute with some JavaScript code to print out a message about the so-called onclick handler. For almost every event type like `click` or `change` or `keypress`, you can put an attribute by prefixing the event name with the word "on". However, **you should never ever use that in real production code!**

Save the above HTML in a file, and run that file in a browser. Open up the browser console (*right-click -> Inspect -> Console*), click on the `<p>` element, and observe what happens. The message "The onclick handler" should appear, then you should see the id `paragraph` printed to the console.

What happened here? The `console.log` shows that an event happened on the `<p>` (i.e. the `event.target`), and yet the `onclick` handler on the `<div>` also fired -- meaning that the click event on the `<p>` bubbled up to the `<div>` and fired its `onclick` event!

Once again, here's the deal:

Don't ever use the `on`-event-name attribute version of an event handler. Instead, always use the `addEventListener` method of the DOM object that

you get from something

like `document.getElementById` or `document.querySelector`.

An event bubbling example

To visualize event bubbling, it might be helpful to watch this short and fun YouTube video on *bubbles inside bubbles inside bubbles*.

Bubble Inside a Bubble Video

We can think of events that happen on nested DOM elements as these nested bubbles. An event that happens on the innermost element bubbles up to its parent element, and that parent's parent element, and so on up the chain. Let's look at another example that demonstrates bubbling.

HTML

```
<!DOCTYPE html>
<html>
  <body>
    <main>
      <div>
        <p>This is a paragraph in a div in a main in a body in an html</p>
      </div>
    </main>

    <script>
      function handler(e) {
        console.log(e.currentTarget.tagName);
      }
      document.querySelector('main').addEventListener('click', handler);
      document.querySelector('div').addEventListener('click', handler);
      document.querySelector('p').addEventListener('click', handler);
    </script>
```

```
</body>
</html>
```

If you save this HTML file, open it in a browser, and click on the `<p>`, three different messages should appear in the console: first “P”, second “DIV”, and third “MAIN”. The click event bubbled upwards from the `<p>` element to the `<div>` and finally to the `<main>`.

We could think of this succession of events as bubbles popping. The innermost bubble (the `<p>` element) *popped* (i.e. displayed an alert), which caused its parent's bubble to pop, which caused its parent's bubble to pop. Since there aren't any `onclick` handlers above the `<main>` nothing else happens on the page, but the bubbles would travel all the way up the DOM until they reached the top (`<html>`) looking for event handlers to run.

Stopping event bubbling with `stopPropagation()`

As stated in the introduction, event bubbling can cause annoying side effects. This MDN doc on [Event bubbling and capture](#) explains what would happen if a user clicked on a `<video>` element that has a parent `<div>` with a show/hide toggle effect. On a click, the video would disappear along with its parent div!

How can you stop this unintended behavior from occurring? The answer is with the `event.stopPropagation()` method which stops the bubbling from continuing up the parent chain. Here's what it would look like on the `<video>` element:

Javascript

```
document
  .querySelector('video')
  .addEventListener('click', event => {
```

```
event.stopPropagation();
video.play();
});
```

Event delegation

While event bubbling can sometimes be annoying, it can also be helpful. The bubbling effect allows us to make use of **event delegation**, which means that we can delegate events to a single element/handler -- a parent element that will handle all events on its children elements.

Say you had an unordered list (``) element filled with several list item (``) elements, and you want to listen for click events on each list item. Instead of attaching a click event listener on each and every list item, you could conveniently attach it to the parent unordered list, like so:

HTML

```
<ul id="my-list">
  <li>This is list item 1.</li>
  <li>This is list item 2.</li>
  <li>This is list item 3.</li>
  <li>This is list item 4.</li>
  <li>This is list item 5.</li>
</ul>
<script>
  document
    .getElementById('my-list')
    .addEventListener('click', e => {
      // will print out "This is list item X"
      // depending on which list item is clicked
      console.log(e.target.innerHTML);

      // always prints "my-list"
```

```
    console.log(e.currentTarget.id);
  });
</script>
```

This example is a lot like the first example you saw with the `<p>` inside of a `<div>`, where the click on the `<p>` bubbled up to the `<div>`. In the above example, a click on any `` will bubble up to its parent, the ``.

When clicked on, a single `` element becomes the `event.target`-- the object that dispatched the event. The `` element is the `event.currentTarget`-- the element to which the event handler has been attached.

Now that you know how to handle events responsibly, go frolic in the bubbles!

What we learned:

- The definition of The Bubbling Principle
- Examples of event bubbling
- How to stop events from bubbling
- How to use bubbling for event delegation