Event Propagation

How are events triggered on elements that are nested inside each other? For example, what events occur if you click on a paragraph inside a section inside the body? What if you have "click" event listeners attached to both the section and to the paragraph? Which one will execute its handler first?

Bubbling and Capturing

```
<script>
// part of our script:
myParagraph.addEventListener("click", foo);
mySection.addEventListener("click", bar);
</script>
...
<body>
  <section>
    <p>Monkey</p>
  </section>
</body>
```

In this example, when a user clicks on the paragraph, the browser emits a MouseEvent that is available to the target element (the paragraph) and *all its parents*. You can think of the MouseEvent as "travelling" from the top of the DOM all the way to the target: it visits the body, then the section, then the paragraph (the target) -- this phase of the MouseEvent is called *capturing*. Then MouseEvent travels all the way back up the DOM: from the paragraph, to the section, then the body -- this phase is called *bubbling*.

(Our example features a MouseEvent, but this is true for all types of events.)

So, to go back to our question: if you have "click" event listeners attached to both the section and to the paragraph, which one will execute its handler first when the paragraph gets clicked? *Answer*: `foo` gets called then `bar` get called. This is because, **by default, event listeners only pay attention to bubbling**, not capturing, so first the paragraph's event listener sees the click event (`foo` gets called), and then the section's event listener sees the event (`bar` gets called).

What if we somehow only clicked the section, but not the paragraph? Then only `bar` would get called.

It works in the same way if the attached handler is the same for both elements:

```
myParagraph.addEventListener("click", foo);
mySection.addEventListener("click", foo);
```

In this case, `foo` would get called twice when the paragraph is clicked: first for the paragraph, then for the section.

What causes capturing to be ignored by default? It's the way we call `addEventListener`. The `addEventListener` method actually accepts three arguments, but the third one is optional and false by default, so we've just been omitting it so far. In other words, writing `addEventListener("click", foo)` is the same as writing `addEventListener("click", foo, false)`, where the third argument means that "paying attention to capturing" is false.

To make an event listener react to capturing in addition to bubbling, call `addEventListener` as follows:

```
// watch only event bubbling
myParagraph.addEventListener("click", foo);
// watch event capturing and bubbling
mySection.addEventListener("click", bar, true);
```

Now when we click the paragraph, first `bar` is called for the section (because of capturing), then `foo` is called for the paragraph (the target), then `bar` is called again for the section (because of bubbling). *Note that old IE has no way to pay attention to event capturing.*

See the available code examples for event object properties that relate to propagation:

- target: the element that triggered the event
- currentTarget: the element to which the handler is attached
- eventPhase: which phase the event is in (capture, bubbling or at target)

One consequence of event propagation is that you have to be careful when adding event listeners to elements that have children: `e.target` may not refer to the element with which the handler was registered.

Event Delegation

We can take advantage of event bubbling to make our JS code less repetitive and more performant by registering just one event handler to a parent element. For example, if all items in a list should react to a click in the same way, then instead of adding a listener to every `li`, just add it to the parent `ul`. You can tell which `li` got clicked by looking at `e.target`.

```
<script>
...
function doSomething(e) {
  //use e.target to refer to the list item that was clicked
  e.target.textContent = "blah";
}
people.addEventListener("click", doSomething);
...
</script>
<body>
  <ul id="people">
    <li>Person 1</li>
    <li>Person 2</li>
    ...
  </ul>
</body>
...
```

This is especially useful if your code might add or remove child elements dynamically, or you don't know in advance how many elements will need an event listener.

Stop Propagation

Sometimes event bubbling clashes with what we want to achieve. Returning to our first example of the paragraph in the section, what if I only want `bar` to be called when just the empty space around the paragraph is clicked? In other words, when the paragraph is clicked, `bar` should not be called.

```
myParagraph.addEventListener("click", foo);
mySection.addEventListener("click", bar);
```

To get that working, I have to stop the MouseEvent from bubbling up to the section from the paragraph. I can do that by adding a call to `stopPropagation` on the event object in `foo`

```
function foo(e) {
  // do paragraph-related stuff
  // then stop the bubbling
  e.stopPropagation();
}
```

Now, when the paragraph is clicked, `foo` will get called, and the event will stop there. `bar` will only get called if we click on an area inside the section that isn't the paragraph.

`stopPropagation` is only available on modern browsers, but there's an alternative for older IE, for example. Cross-browser solution:

```
if (e.stopPropogation) { // W3C/addEventListener()
  e.stopPropogation();
} else { // Older IE.
  e.cancelBubble = true;
}
```

Last update: 2017-02-25 22:33:22

Source: _javascript_423/lectures/06_event_propagation.md