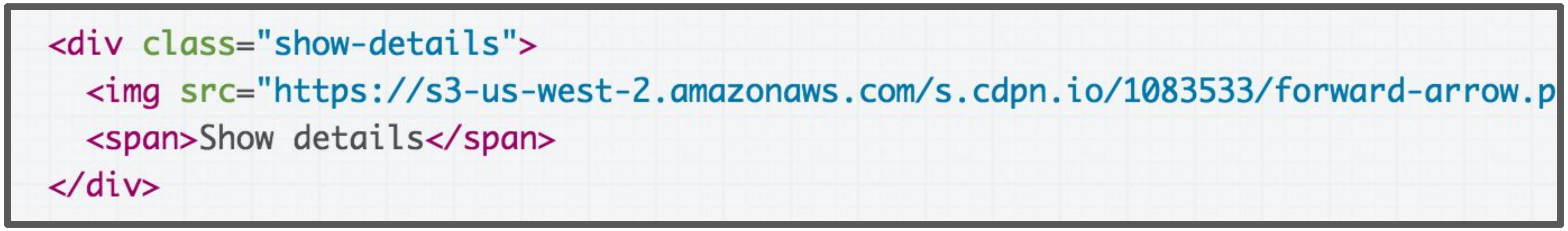
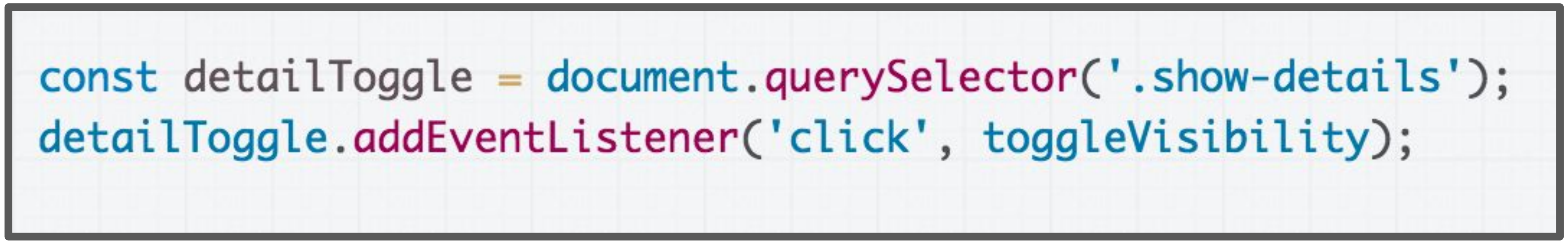
JavaScript events in detail

Events in JavaScript

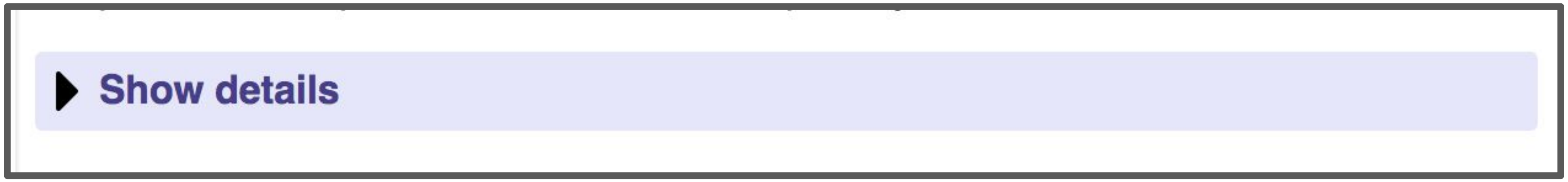
If you put a "click" event listener on an element, what happens if the user clicks a *child* of that element?



```html
<div class="show-details">
  <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/forward-arrow.p
  <span>Show details</span>
</div>
```

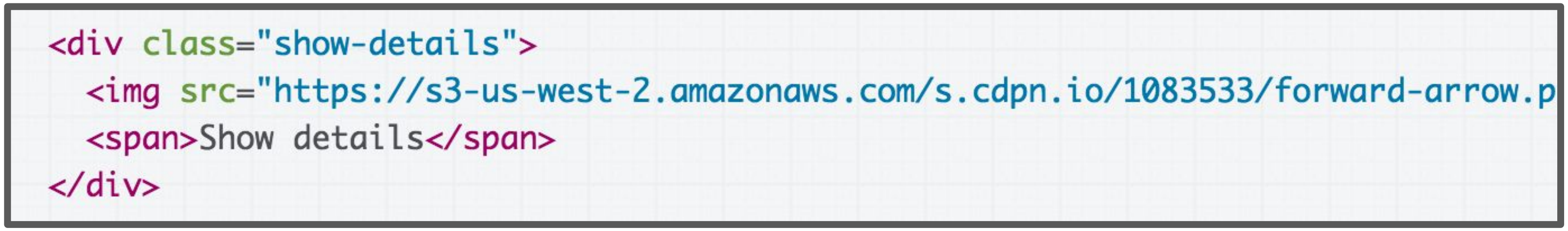


```javascript
const detailToggle = document.querySelector('.show-details');
detailToggle.addEventListener('click', toggleVisibility);
```

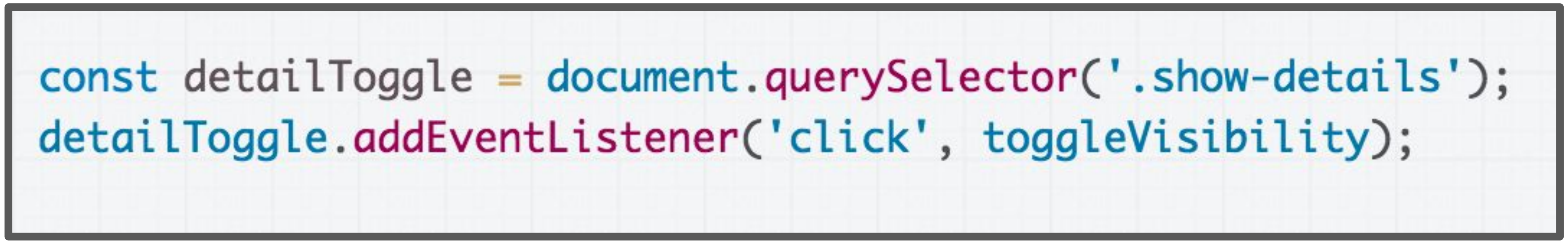


Events in JavaScript

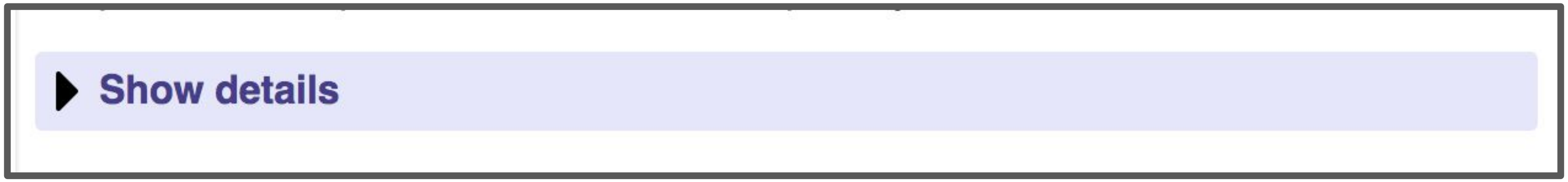
Example: If you click on the `<img>`, will the `toggleVisibility` function fire?



```html
<div class="show-details">
  <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/forward-arrow.p
  <span>Show details</span>
</div>
```



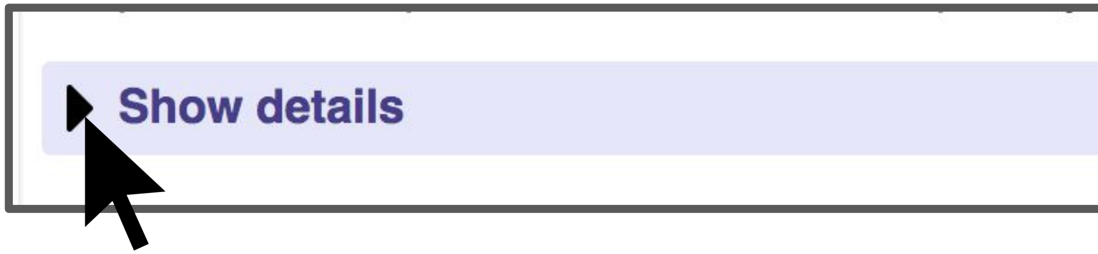
```javascript
const detailToggle = document.querySelector('.show-details');
detailToggle.addEventListener('click', toggleVisibility);
```

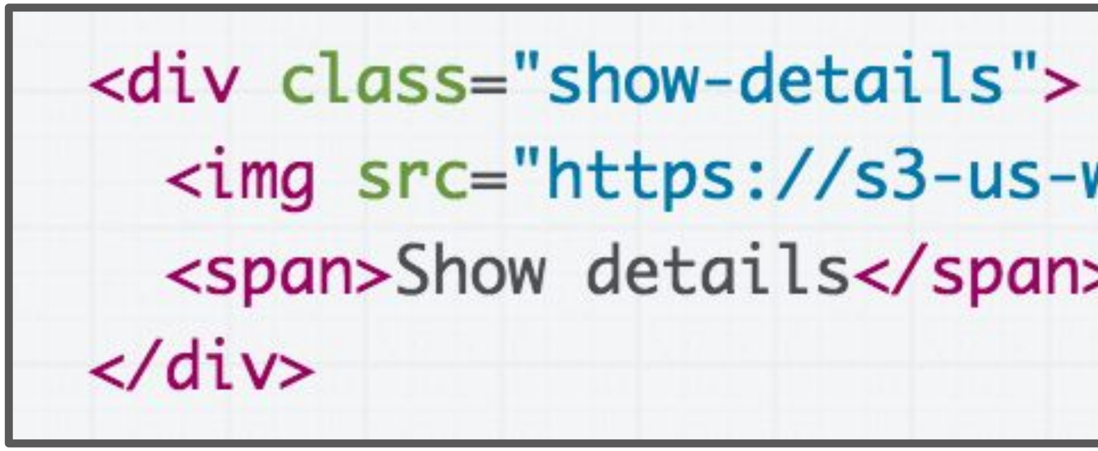


Events in JavaScript

Yes, a click event set on an element will fire if you click on a child of that element

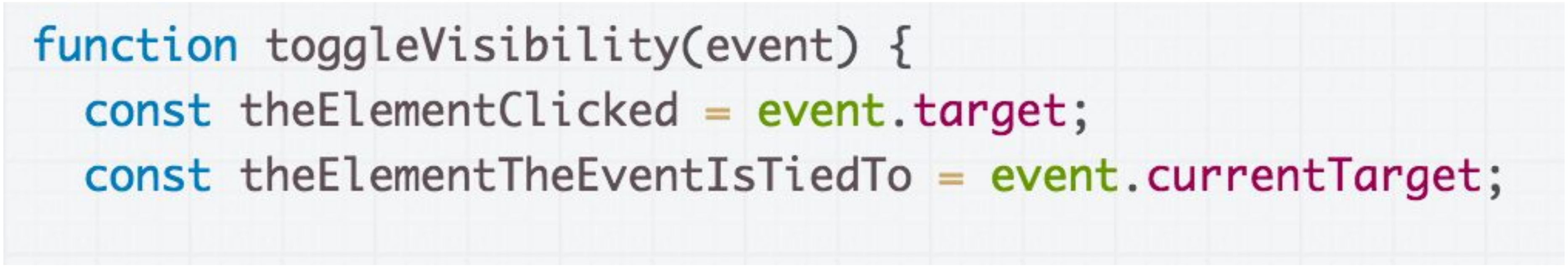
If you put a click event listener on the `div`, and the user clicks on the `img` inside that `div`, then the event listener will still fire.





```
<div class="show-details">
  <img src="https://s3-us-w
  <span>Show details</span
</div>
```

Event.currentTarget vs target



```
function toggleVisibility(event) {
  const theElementClicked = event.target;
  const theElementTheEventIsTiedTo = event.currentTarget;
```

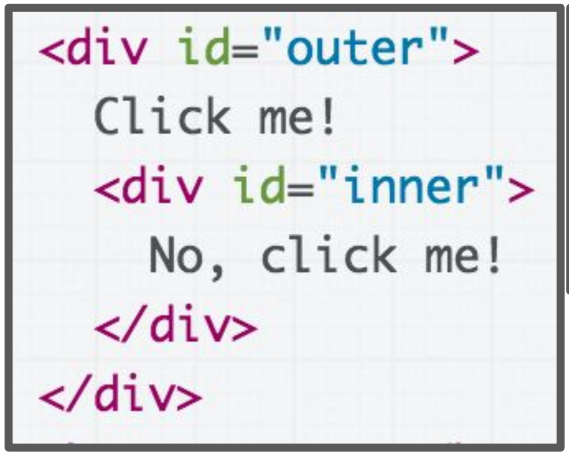
You can access either the element clicked or the element to which the event listener was attached:

- ***event*.`target`**: the element that was clicked / "dispatched the event" (might be a child of the target)
- ***event*.`currentTarget`**: the element that the original event handler was attached to

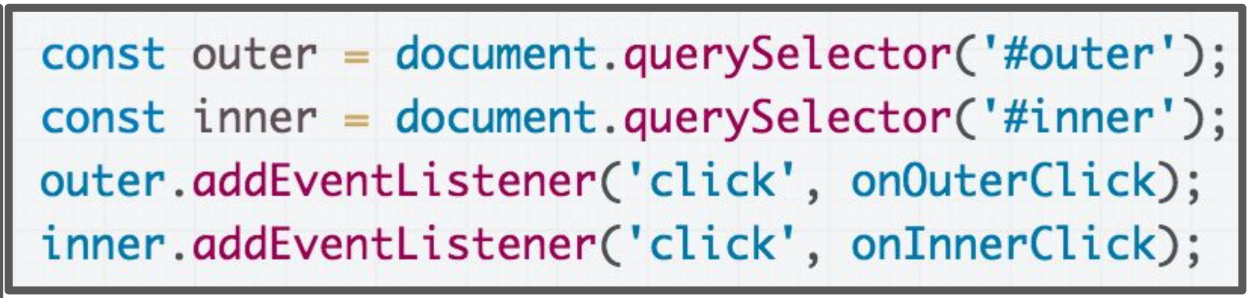
Multiple event listeners

What if you have event listeners set on both an element and a child of that element?

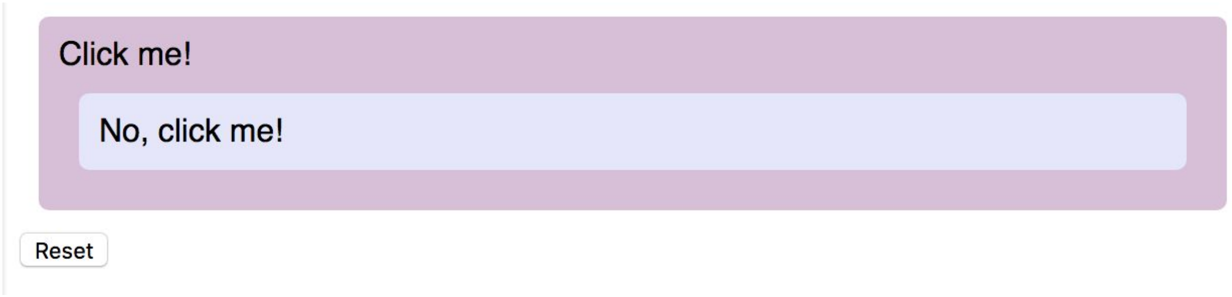
- Do both fire?
- Which fires first?



```html
<div id="outer">
  Click me!
  <div id="inner">
    No, click me!
  </div>
</div>
```



```js
const outer = document.querySelector('#outer');
const inner = document.querySelector('#inner');
outer.addEventListener('click', onOuterClick);
inner.addEventListener('click', onInnerClick);
```



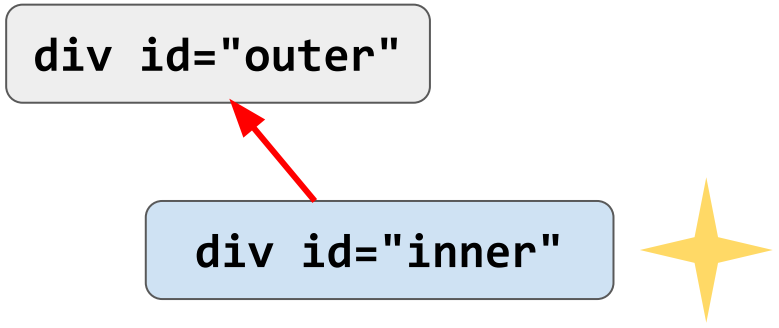
Click me!

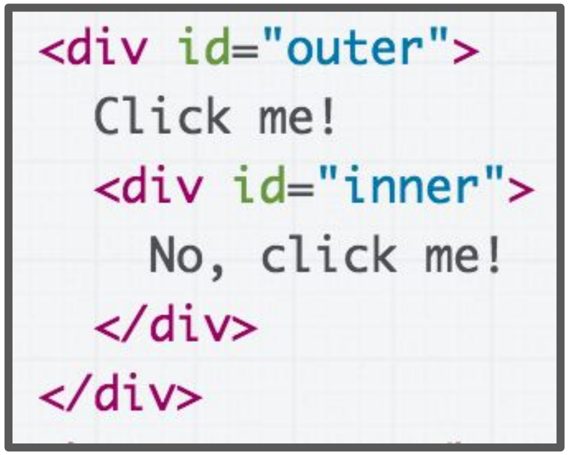
No, click me!

(CodePen)

Event bubbling

- Both events fire if you click the inner element
- By default, the event listener on the inner-most element fires first



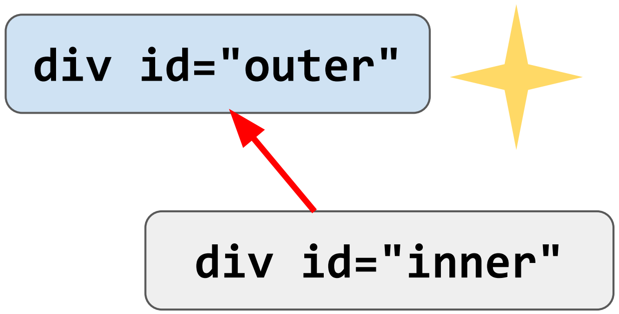


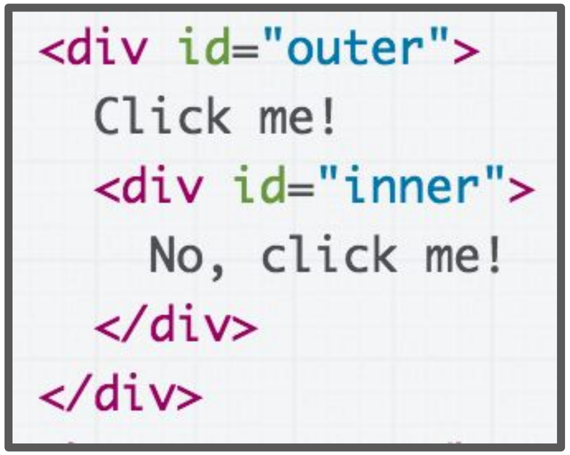
```
<div id="outer">
  Click me!
  <div id="inner">
    No, click me!
  </div>
</div>
```

This event ordering (inner-most to outer-most) is known as **bubbling**. (CodePen)

Event bubbling

- Both events fire if you click the inner element
- By default, the event listener on the inner-most element fires first



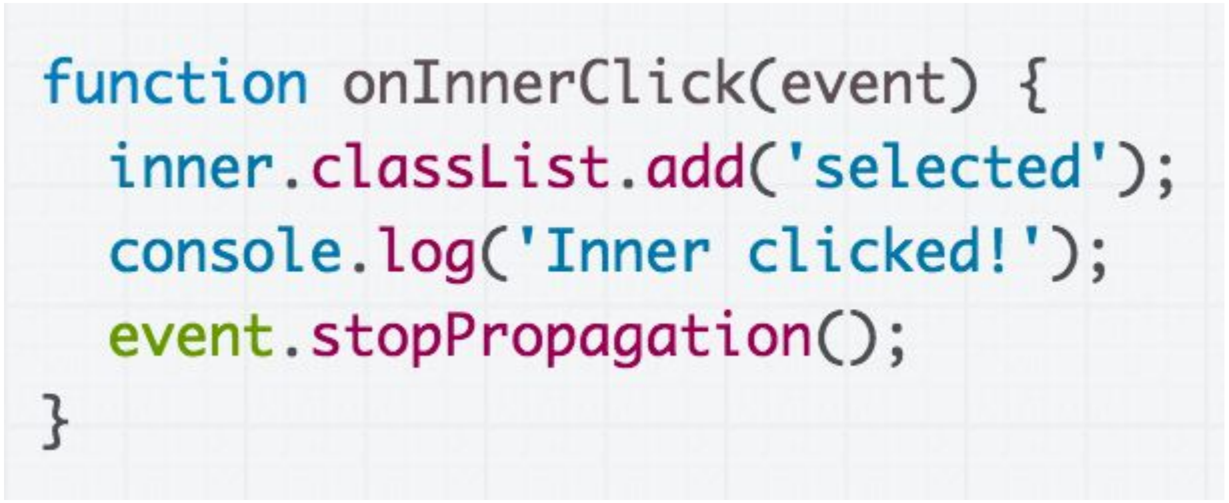


```
<div id="outer">
  Click me!
  <div id="inner">
    No, click me!
  </div>
</div>
```

This event ordering (inner-most to outer-most) is known as **bubbling**. (CodePen)

stopPropagation()

We can stop the event from bubbling up the chain of ancestors by using ***event***`.stopPropagation()`:



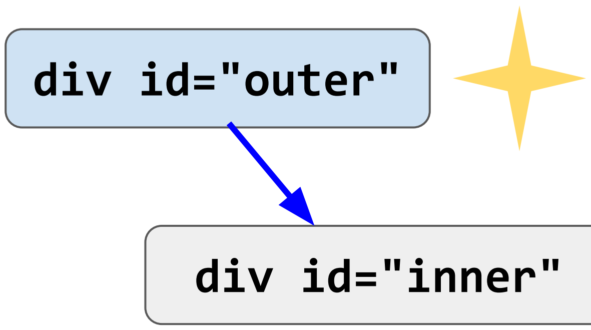
```
function onInnerClick(event) {
  inner.classList.add('selected');
  console.log('Inner clicked!');
  event.stopPropagation();
}
```

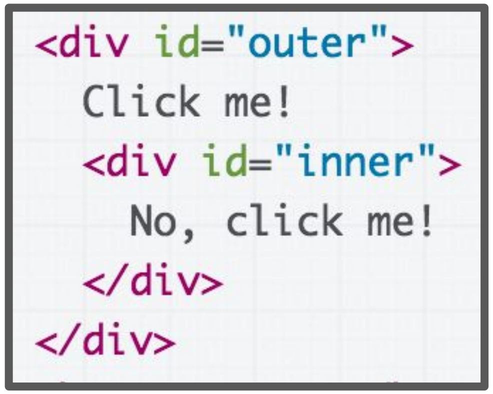
See [default behavior]() vs with [`stopPropagation`]()

Event capturing

To make event propagation go the opposite direction, add a 3rd parameter to `addEventListener`:

```
event.addEventListener(
    'click', onClick, { capture: true} );
```





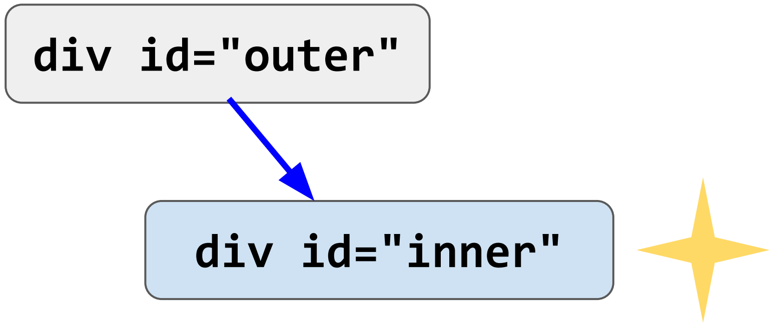
```
<div id="outer">
  Click me!
  <div id="inner">
    No, click me!
  </div>
</div>
```

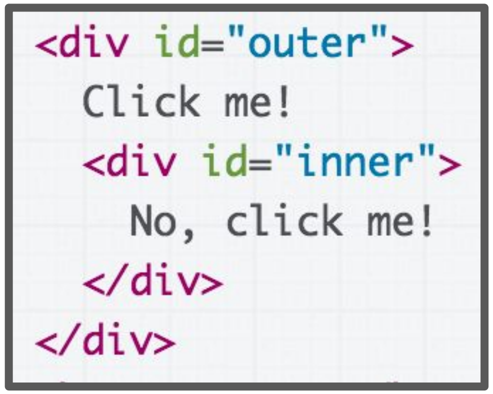
This event ordering (outer-most to inner-most) is known as **capturing**. (CodePen)

Event capturing

To make event propagation go the opposite direction, add a 3rd parameter to `addEventListener`:

```
event.addEventListener(
    'click', onClick, { capture: true} );
```



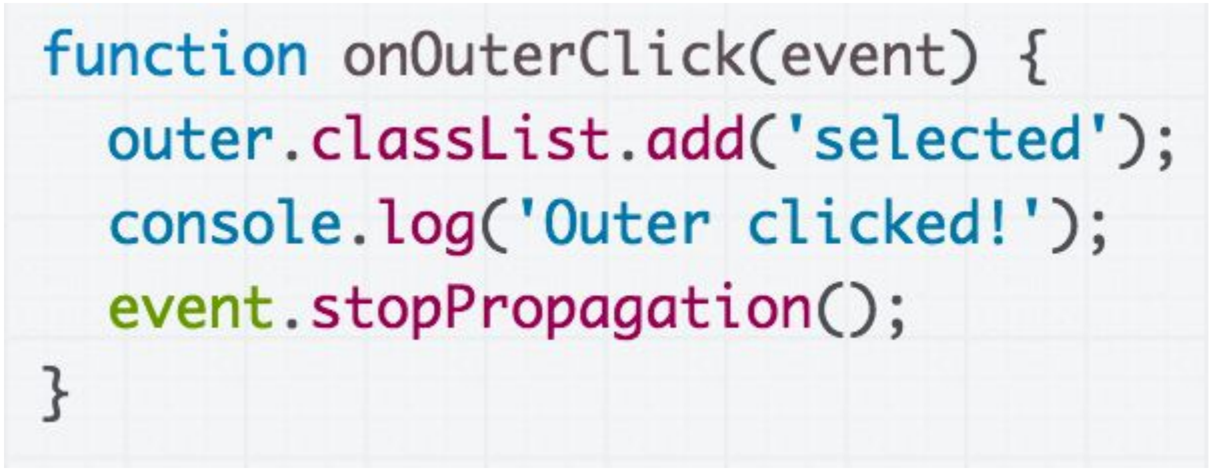


```
<div id="outer">
  Click me!
  <div id="inner">
    No, click me!
  </div>
</div>
```

This event ordering (outer-most to inner-most) is known as **capturing**. (CodePen)

stopPropagation()

We can also use ***event***`.stopPropagation()` in capture-order:



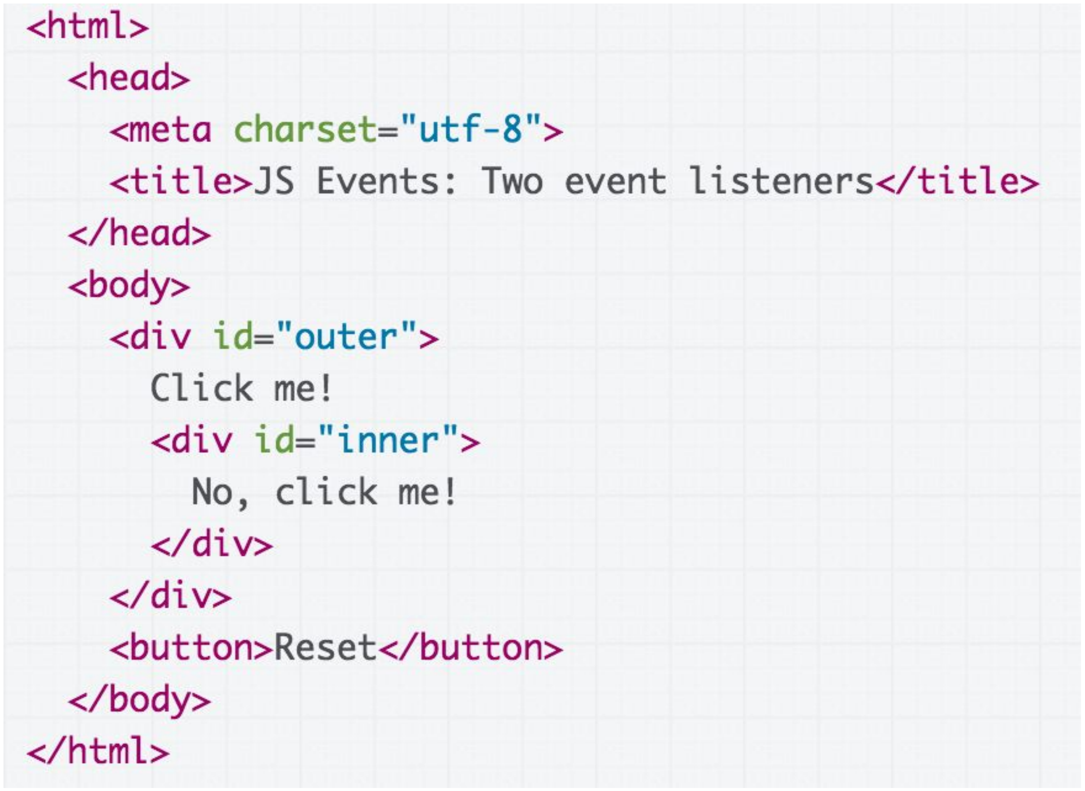
```
function onOuterClick(event) {
  outer.classList.add('selected');
  console.log('Outer clicked!');
  event.stopPropagation();
}
```

See default behavior vs with `stopPropagation`

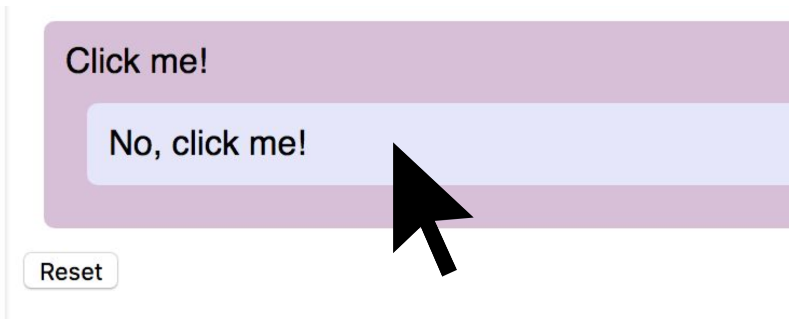
Some technical details...

Behind the scenes

Technically, the browser will go through **both** a capture phase and a bubbling phase when an event occurs:



```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event listeners</title>
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
</html>
```

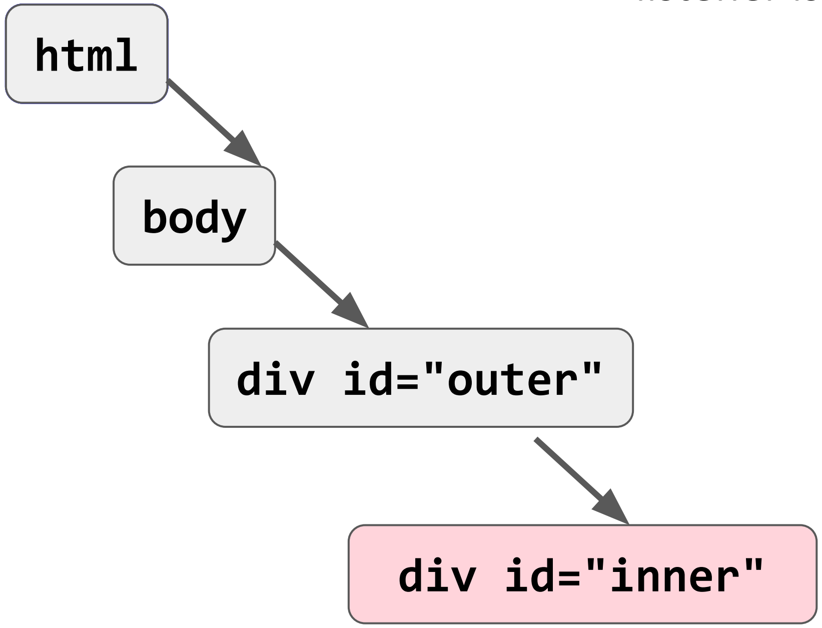


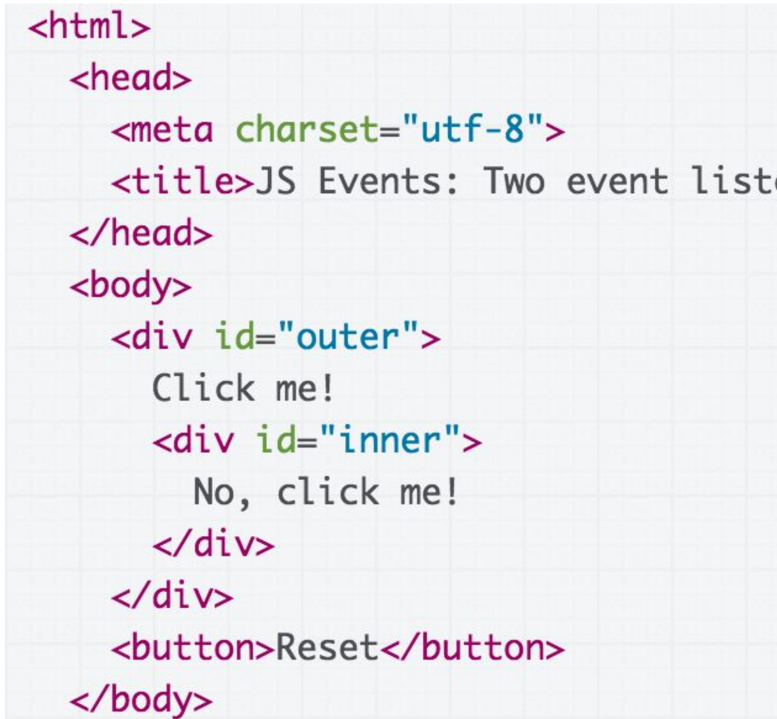
If we click on the div with id="inner"...

Behind the scenes

The browser creates the `target`'s"**propagation path**," or the list of its ancestors up to root (w3c)

(`target` meaning the thing you clicked; not necessarily the element the event listener is attached to)

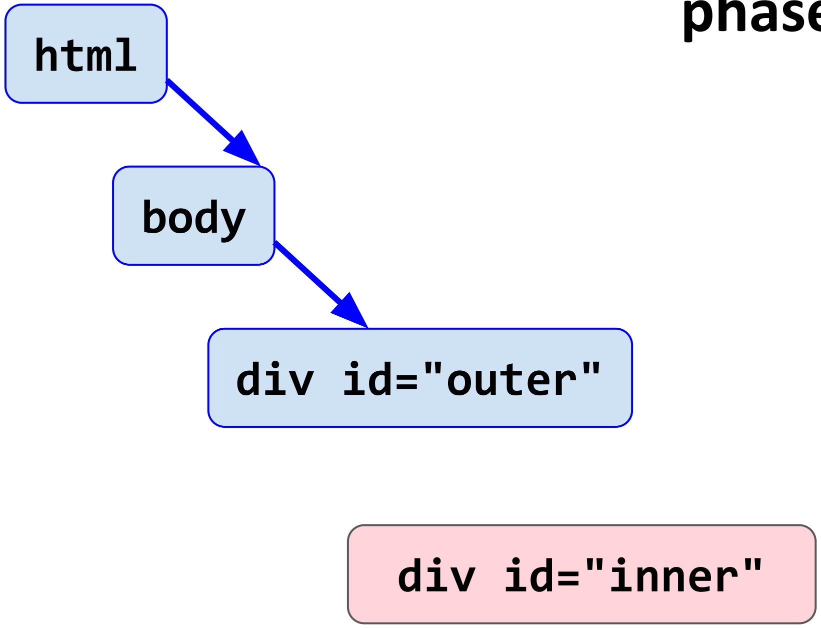


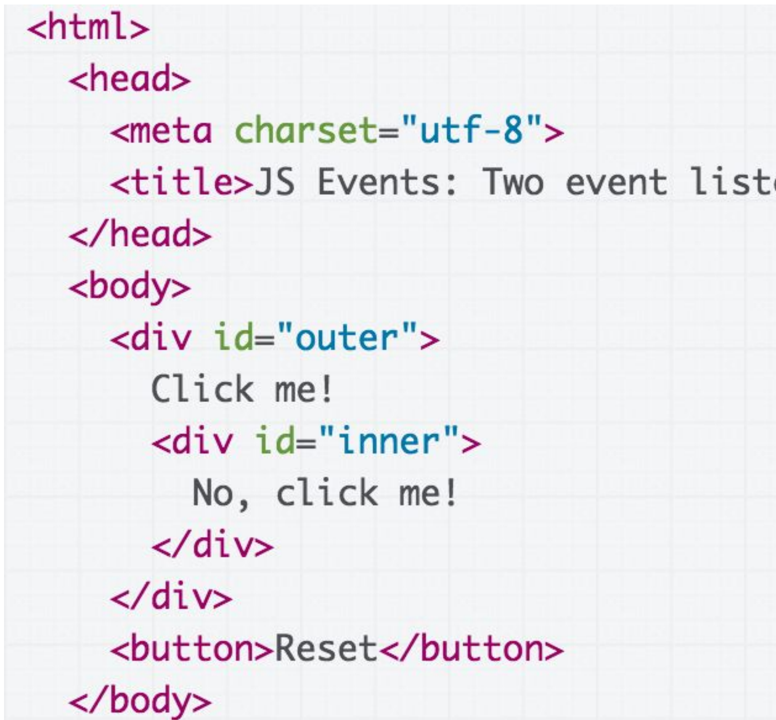


```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Capture phase"

The browser begins at the top of the propagation path and invokes any event listeners that have `capture="true"`, in path order until it gets to the target. This is the "**capture phase**" (w3c)

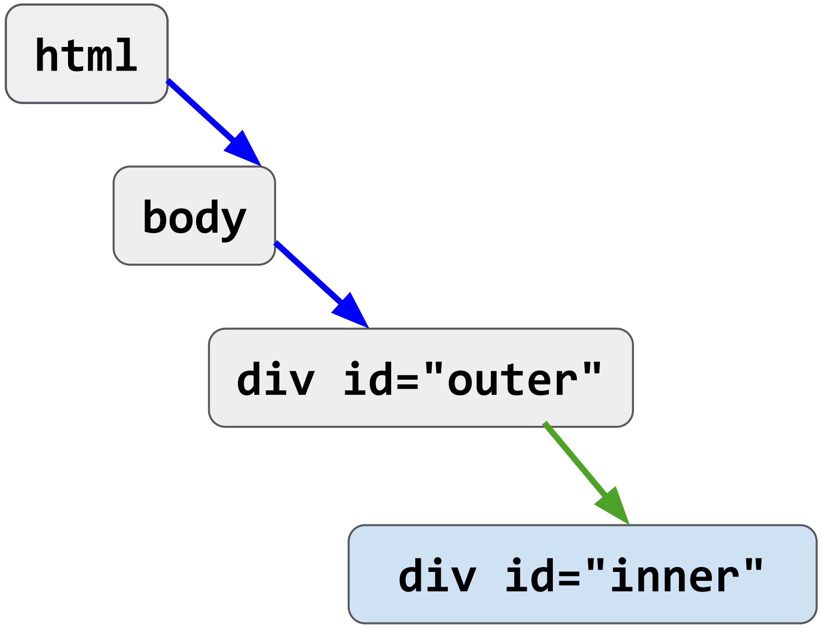


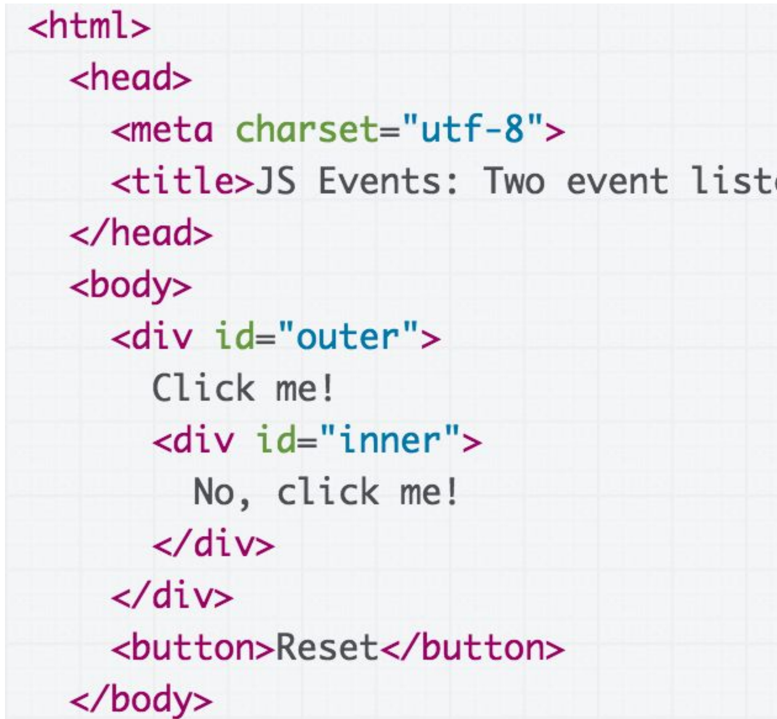


```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Target phase"

Then the browser invokes any event listener that was set on the target itself. This is the "target phase" (w3c)

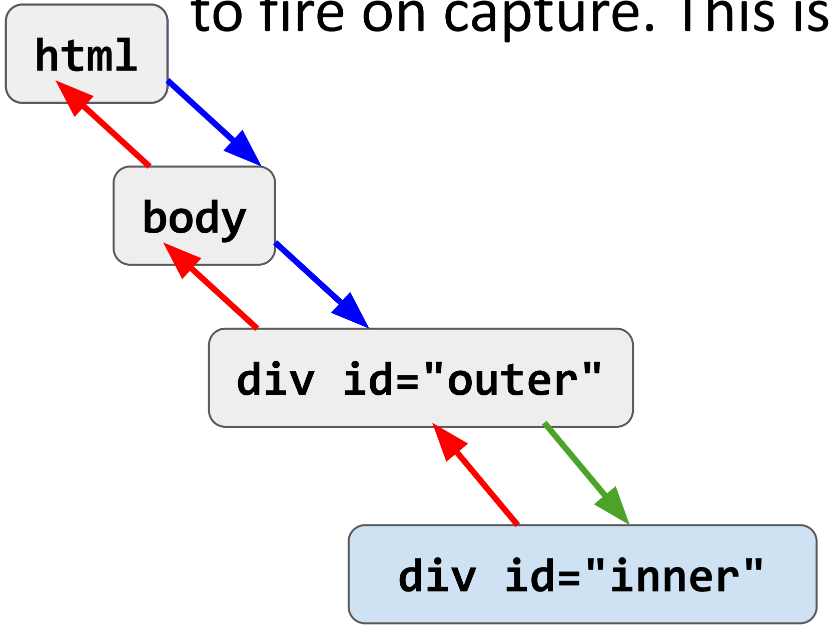


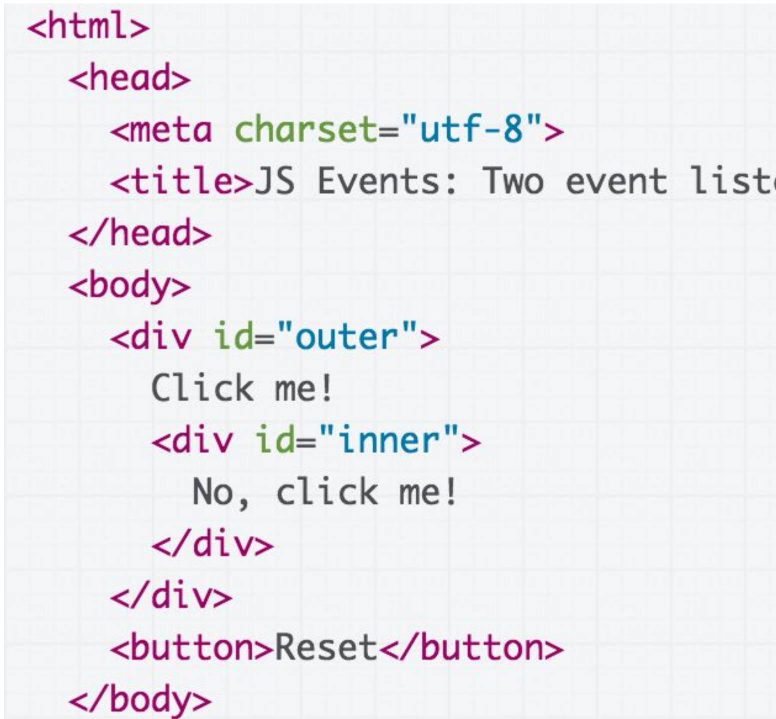


```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Bubble phase"

If the event type has `bubbles=true` (see `click`, e.g.) the browser goes back up the propagation path in reverse order and invokes any event listener that wasn't supposed to fire on capture. This is the "**bubble phase**" (w3c)

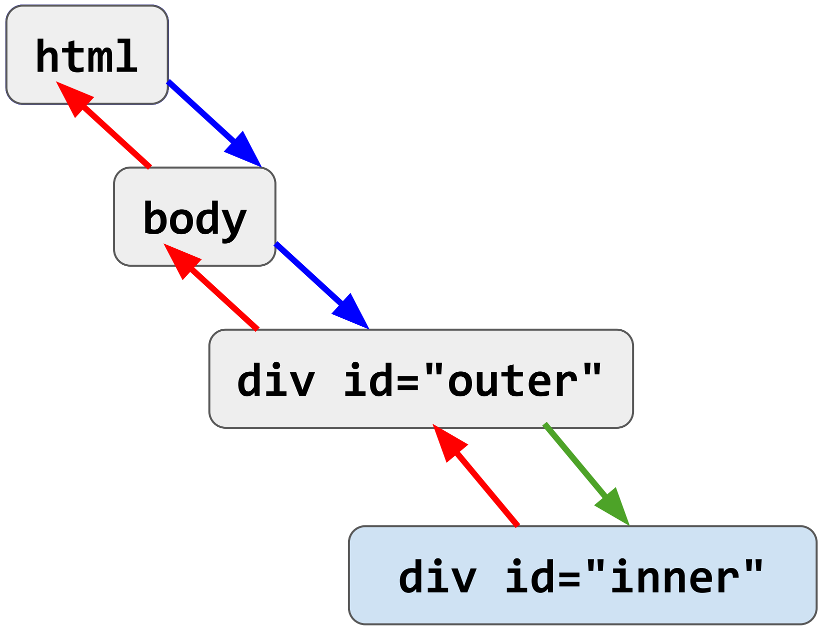




```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

stopPropagation()

Therefore `stopPropagation()` actually stops the rest of the 3-phase dispatch from executing



In Practice

Don't worry about:

- You never need to use capture order - you can always use bubbling
- You don't really need to know how the browser goes through "capture phase", "target phase", then "bubble phase"

Do worry about:

- **You do need to understand bubbling, though**
- `stopPropagation()` also comes in handy