EVENT LISTENERS AND HANDLERS

Listen up

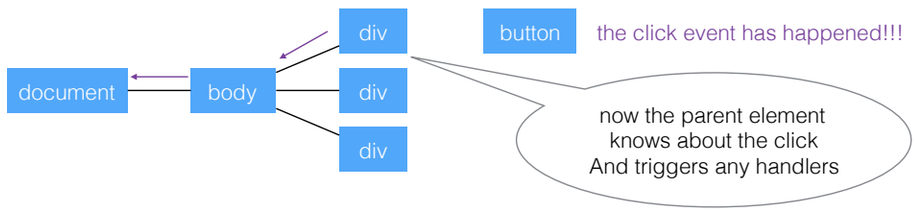
EVENT HANDLERS

```
element.addEventListener('click', function (event) {
    // Run this code on click
});
```

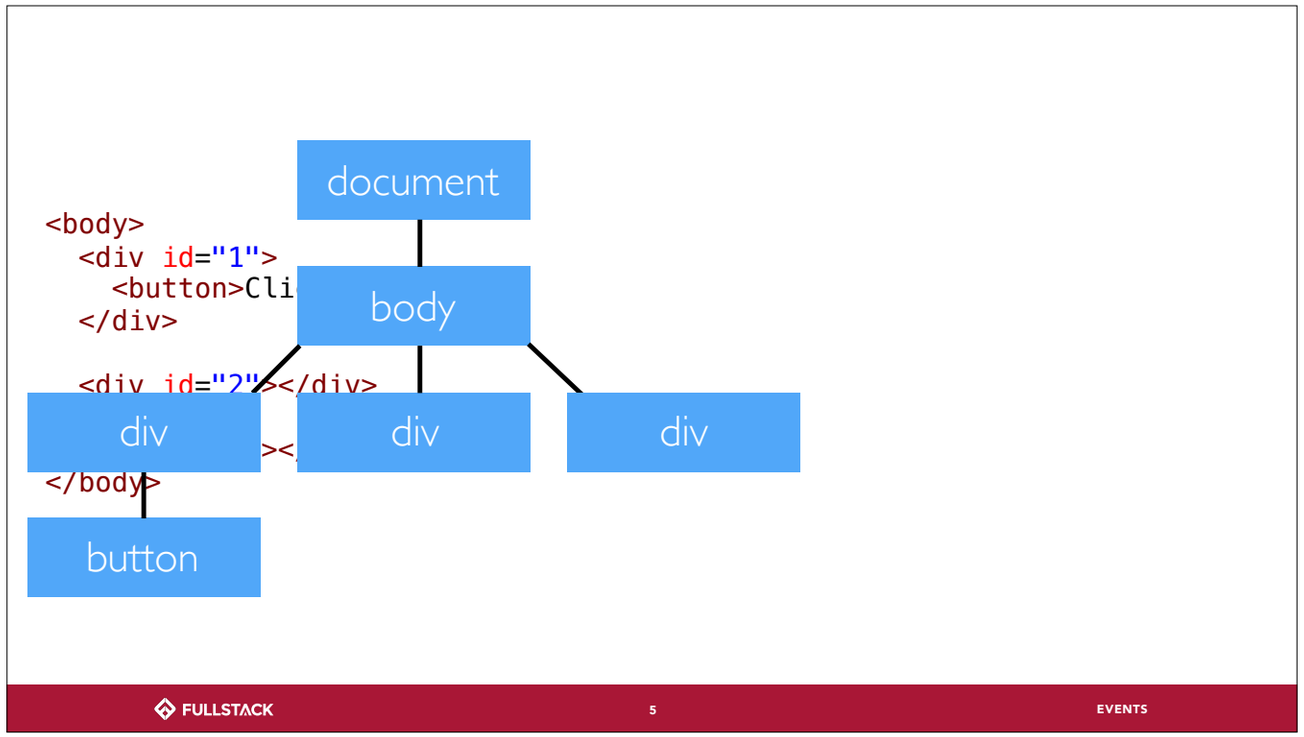
- JS that handles things that happen in the DOM
- Event examples:
  - click
  - (form) submit
  - mouseover
  - scroll

EVENT PROPAGATION/BUBBLING

- An event is directed to its intended target
- If there is an event handler it is triggered
- From here, the **event** *bubbles* up to the containing elements
- This continues to the document element itself



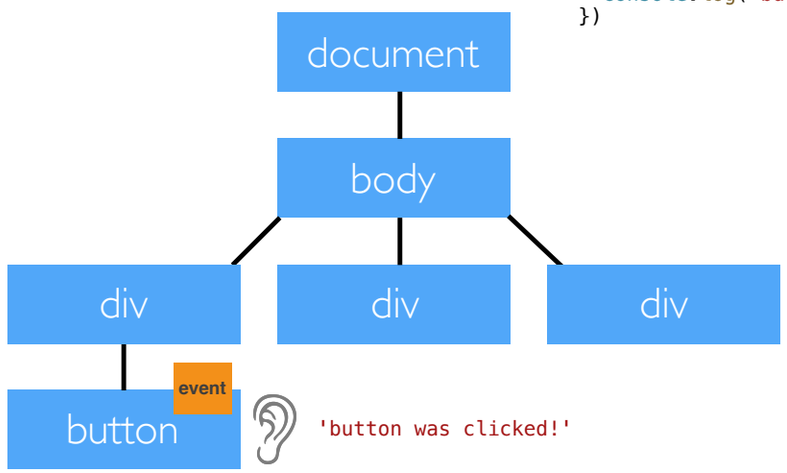
```
<body>
  <div id="1">
    <button>Click Me</button>
  </div>

  <div id="2"></div>

  <div id="3"></div>
</body>
```

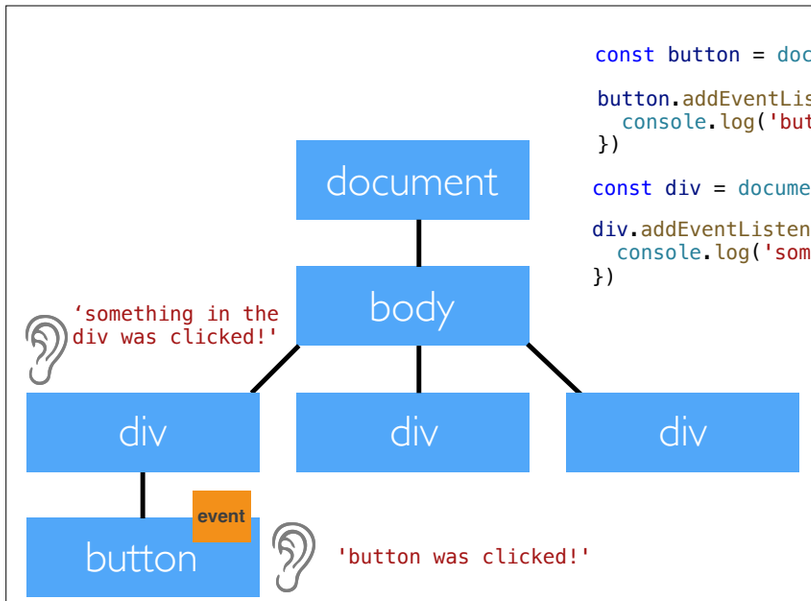

document
<body>

<button>Cli

</body>
body
div
div
div
button

```
const button = document.getElementsByTagName('button')[0]

button.addEventListener('click', function (evt) {
  console.log('button was clicked!')
})
```

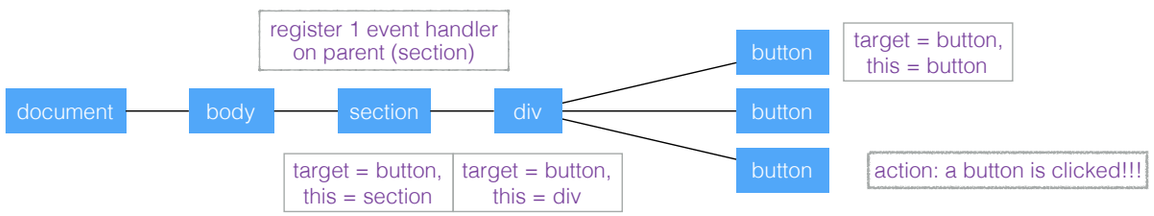





```
const button = document.getElementsByTagName('button')[0]

button.addEventListener('click', function (evt) {
  console.log('button was clicked!')
})

const div = document.getElementById('one')

div.addEventListener('click', function (evt) {
  console.log('something in the div was clicked!')
})
```

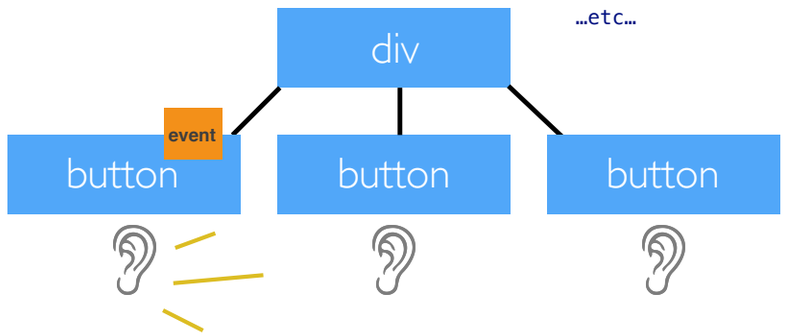
EVENT DELEGATION

- The process of using event propagation to handle events at a higher level in the DOM
- Allows for a single event listener



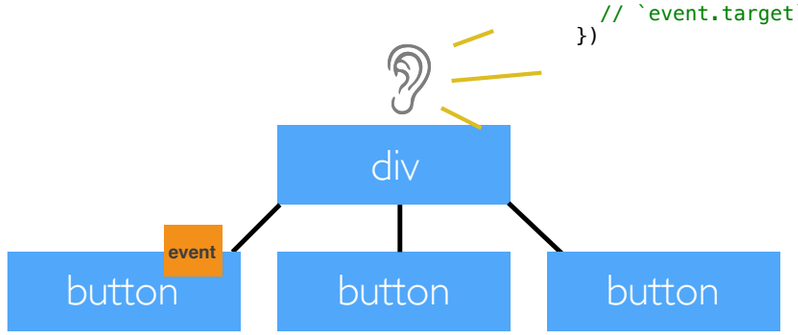
Without Event Delegation

```
const button1 = document.getElementsByTagName(‘button')[0]
const button2 = document.getElementsByTagName(‘button')[1]
const button3 = document.getElementsByTagName(‘button')[2]

button1.addEventListener('click', function (event) {
})

button2.addEventListener(‘click', function (event) {
})

…etc…
```



With Event Delegation

```
const div = document.getElementById('button-container')

div.addEventListener('click', function (event) {
  // `this` -> div
  // `event.target` -> button
})
```



THIS

THIS

- …is the “context” for a function.
- …is determined when a function is *invoked*, not when it is defined (**exception**: arrow functions).

  To determine what `this` is for any function, take a look at its *call-site*.

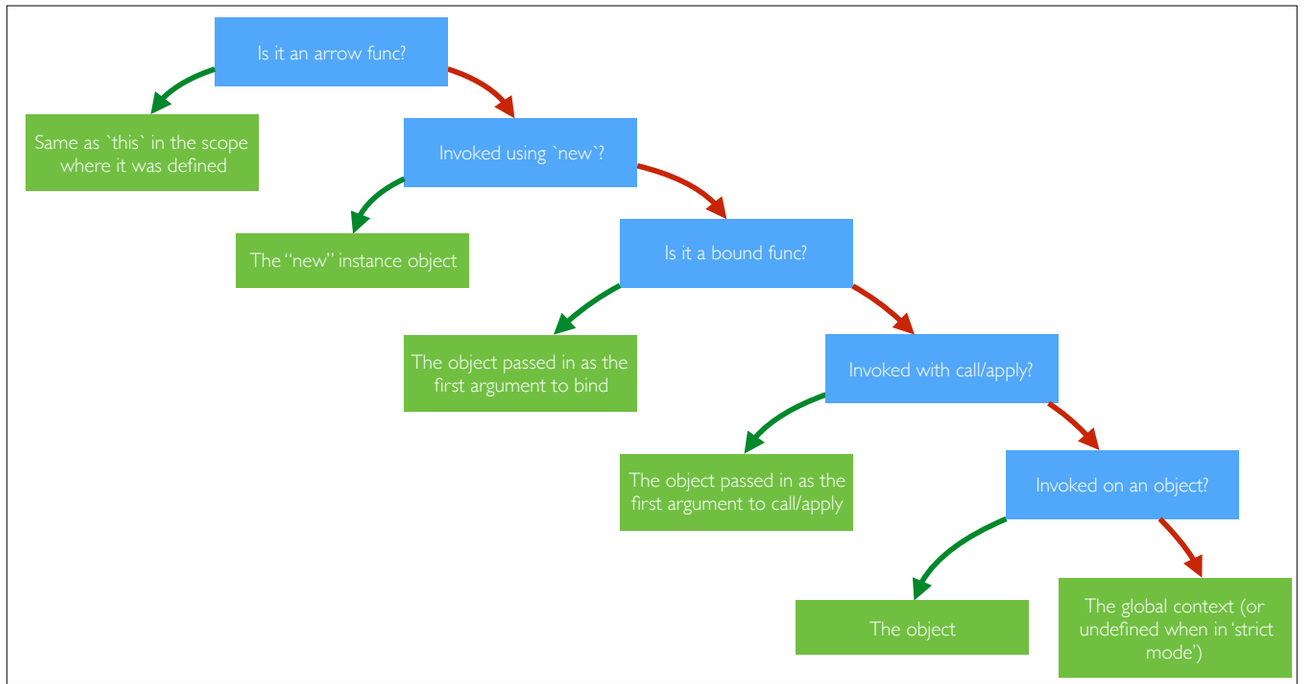
TYPES OF CONTEXT BINDING AND CALL-SITE

- Default binding: `func();`
- Implicit binding: `obj.func();`
- Explicit binding: `func.call(obj);`
- “new” binding: `new func();`

THE .BIND METHOD

- Requires one argument, a `thisArg`.
- Returns a new function whose `this` is always the thisArg.
- Does *not* invoke the function.

```
const boundFunc = oldFunc.bind(thisArg);
boundFunc(); //invoked with thisArg as `this`
```


Is it an arrow func?
Same as `this` in the scope where it was defined
Invoked using `new`?
The "new" instance object
Is it a bound func?
The object passed in as the first argument to bind
Invoked with call/apply?
The object passed in as the first argument to call/apply
Invoked on an object?
The object
The global context (or undefined when in 'strict mode')

WORKSHOP TIME