POWERING POTENTIAL

Events

JavaScript Fundamentals

Introduction

- Understanding JavaScript events
- Subscription models
  - Inline
  - Programmatic
  - Event listeners
- Event bubbling and capturing
- The Event object
- The ‘this’ keyword

Understanding JavaScript events

Events are the beating heart of any JavaScript page

- JavaScript was designed to provide interactivity to web pages
- This means our pages become responsive to users

Events can be tricky as older browsers implemented them badly

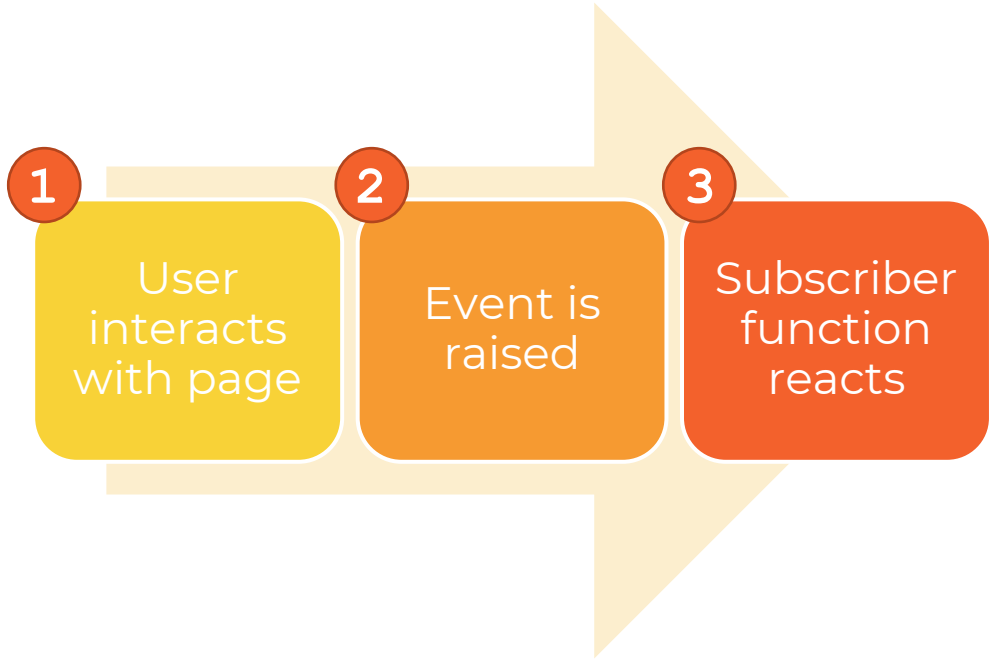
Can be implemented as hardcoded attributes or programmatically

- Inline hardcoded will work everywhere, but can be a blunt instrument
  - Always on, always do the same thing
- Programmatic events are reusable and can be more sophisticated
  - Conditional events depending on browser
  - Detachable - can be switched off

The JavaScript event model

The JavaScript event model uses a publisher/subscriber model.

- Event is raised, a DOM raises countless events
- If a function has been subscribed to the event, it fires



The inline subscription model

- The inline subscription model hardcodes events in the HTML
  - It's quick, easy and works in all browsers
- This approach is okay for testing but not recommended for release
  - It will likely lead to hard to maintain and repetitive code
  - The event is always on and always fires
- Different events models may be needed for different UI

```
<button type="button" onclick="changeClass('container', 'div2');">
```

- Choose the event you wish to subscribe to
  - Add function call code as the attribute value

Simple event registration model

- All modern browsers accept this programmatic registration approach
  - Events are properties of DOM objects
  - You can assign an event to a function

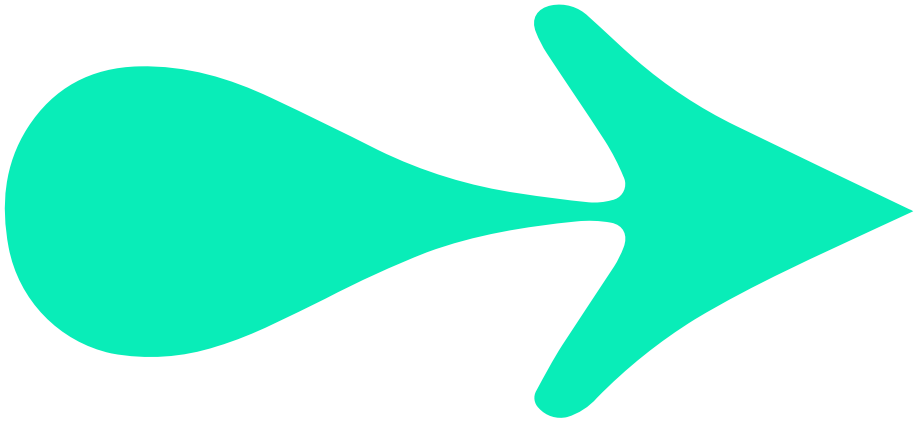
```
myObject.onclick = functionName;
```

- This can also be achieved with anonymous functions
  - Very useful when you only want one object to raise the function

```
myObject.onclick = function(){
    //code to do stuff
}
```

- This approach limits one event to one behaviour, unless you use nested function calls

Event listener registration model



- Allows multiple subscribers to the same events
- Can be detached easily during the life of the program
- Event listeners can be added to any DOM event
  - DOM object selected as usual
  - **`addEventListener`** method setup takes three parameters:
    1. The event
    2. The function to be raised
    3. Whether event capturing should occur (optional, **`default: false`**)

Using addEventListener

- Using **`addEventListener`** is quite simple
  - Parameter 1 – is the event
  - Parameter 2 – is the function
  - Parameter 3 – is a Boolean event bubbling property

```
let e = document.getElementById('container')
e.addEventListener('click', callMe, false);
```

- Multiple events can be subscribed to the same element

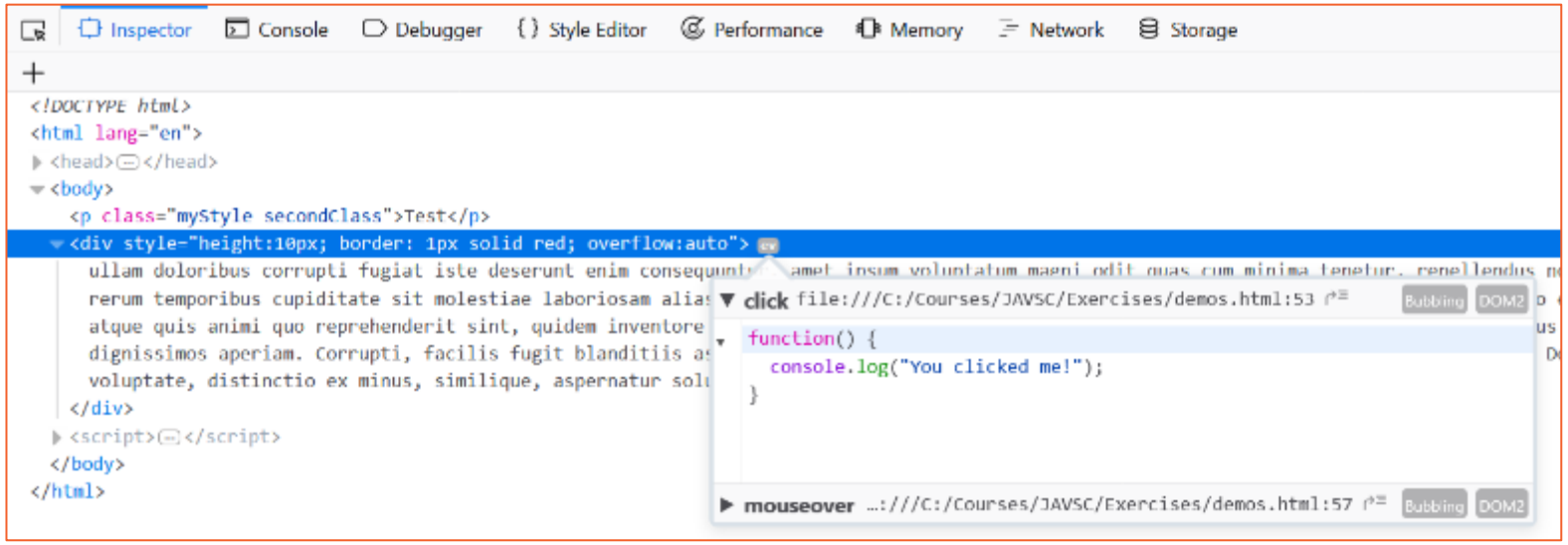
```
e.addEventListener('click', callMe, false);
e.addEventListener('click', alsoCallMe);
```

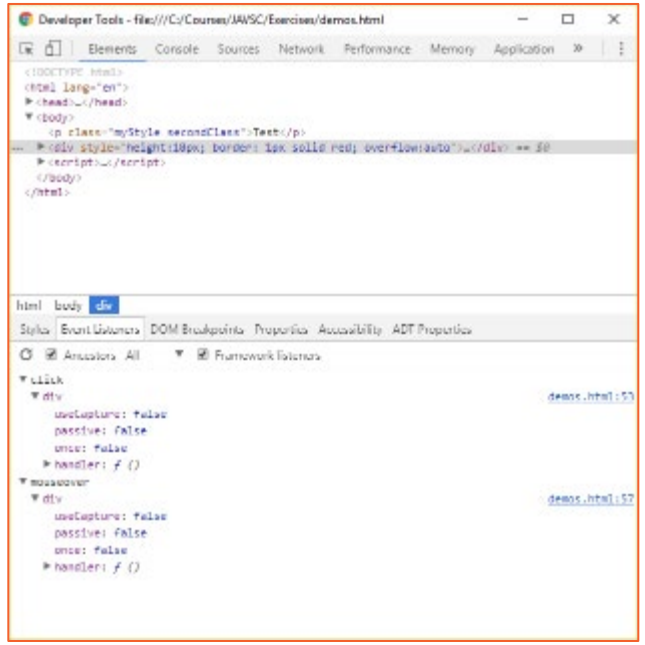
Debugging event listeners

Previously, we have examined a DOM object to see the event.

- This will not work with event listeners
- The ‘hooking up’ occurs behind the scenes

Browser debug tools can help us out here.





addEventListener and anonymous functions

In many situations, we want to conceal event-raising functions.

- Sounds like a job for anonymous functions!

```
e.addEventListener('click', function () { alert('Do stuff'); });
```

By using the **addEventListener** approach, no parameters can be passed.

- With the exception of the event object (covered later)
- We can get around this issue with anonymous functions

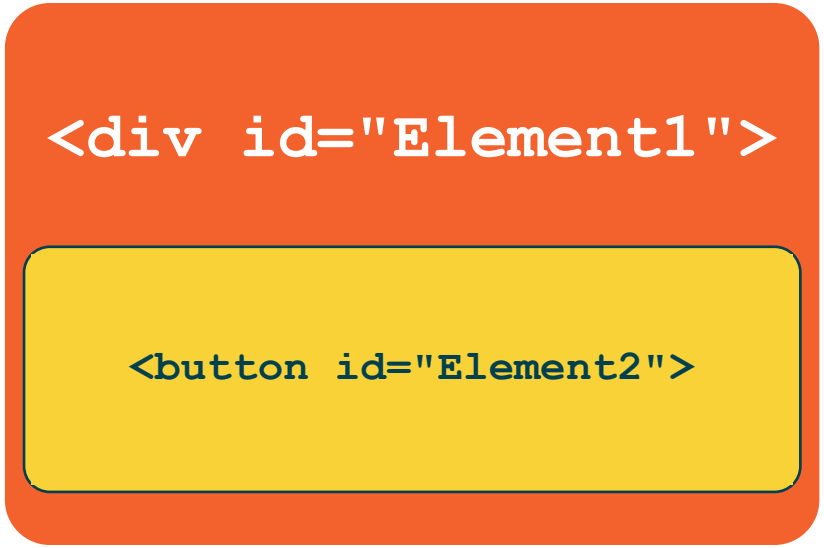
```
b.addEventListener('click', function () {
changeClass(e, 'div2');

});
```

Nested function call

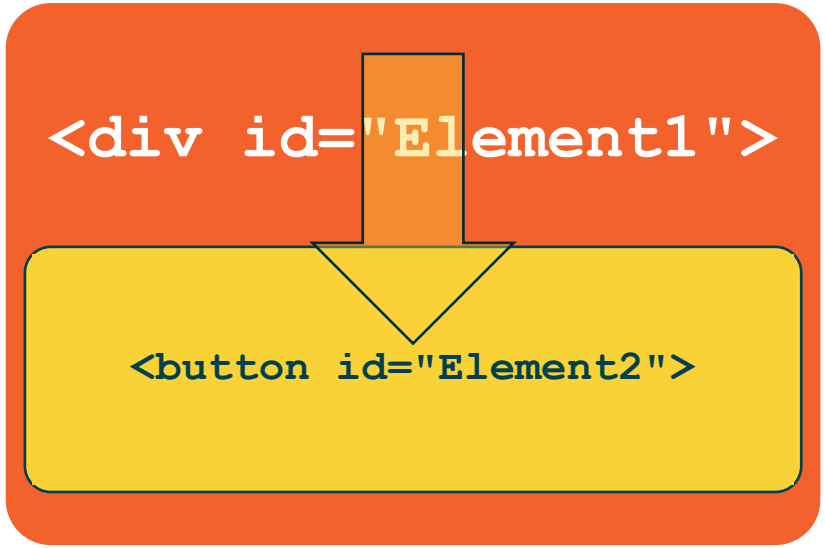
Event Bubbling vs Capturing

- Events in JavaScript can bubble or be caught
  - W3C browsers implement both
  - IE8 or less only implements bubbling
- Consider the problem of an element nested inside another element
  - (e.g., button in a div)
  - Both have an onclick event
- If the user clicks on Element2, they implicitly click on Element1
  - Which event handler should fire first?
  - This is what bubbling and capture define



Event Capturing

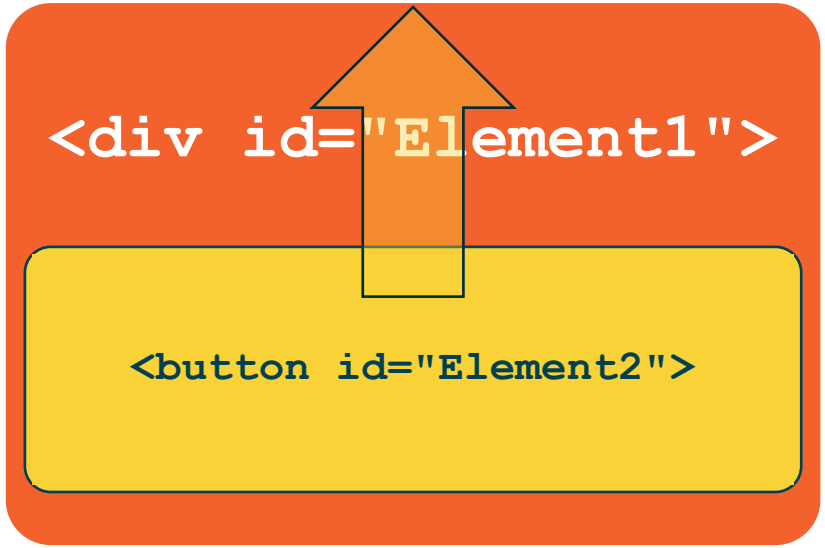
- With Event Capturing, the event handler of Element1 fires first



- You are very unlikely to ever encounter a capture only browser.

Event Bubbling

- With Event Bubbling, the event handler of Element2 fires first

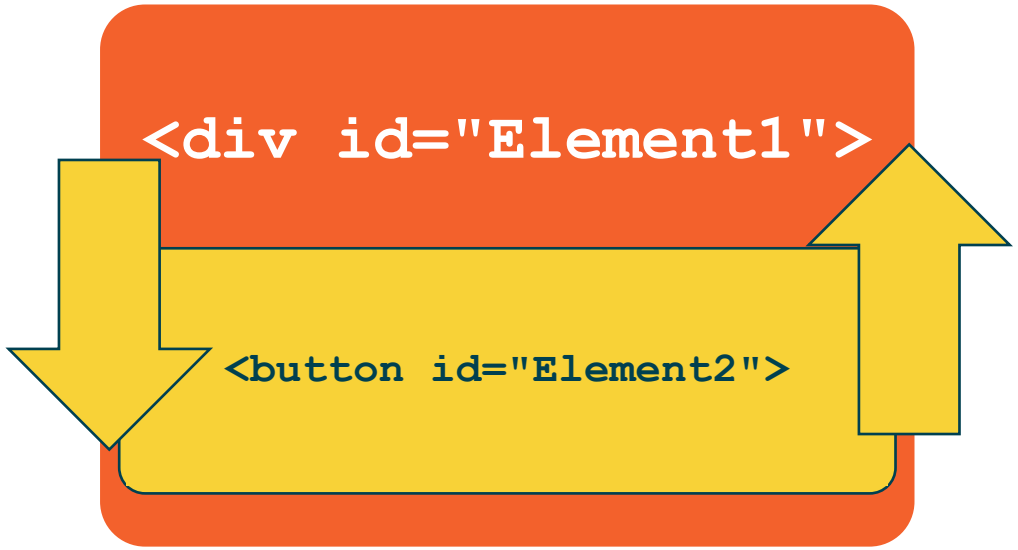


- Legacy Note: IE8 and below could only use bubbling

W3C model

The W3C model can use either approach:

- The third parameter in **`addEventListener`** sets how the event works
- You can mix both in the same page, if appropriate



Removing Event Listeners

- Done using the `removeEventListener()` function
- Arguments must be exactly the same as the arguments used to add the event in the first place
  - Event type must be the same
  - Event handler function must be the same
  - Any options, including bubbling and capturing, must be the same

The event object (1)

You create an event object whenever you raise an event.

- With any of the approaches we have taken

The event has a series of useful properties for us to consider.

Property	Description
bubbles	Returns whether or not an event is a bubbling event.
cancelable	Returns whether or not an event can have its default action prevented.
currentTarget	Returns the element whose event listeners triggered the event.
target	Returns the element that triggered the event.
timeStamp	Returns the time (in milliseconds) at which the event was created.
type	Returns the name of the event.

The event object (2)

The event object is created and passed when an event occurs.

- You can add a parameter to the event handling function to catch it

```
a1.addEventListener('click', stopDefault);
function stopDefault(evt) {
    evt.preventDefault();
    evt.stopPropagation();
}
```

Prevents event bubbling

Stops the hyperlink from redirecting

Different kinds of events allow you to capture additional information.

- Such as, key pressed or mouse button clicked

```
b.addEventListener('mousedown', mouseEvent, true);
function mouseEvent(e) {
    alert(`${e.pageX} ${e.pageY}`);
}
```

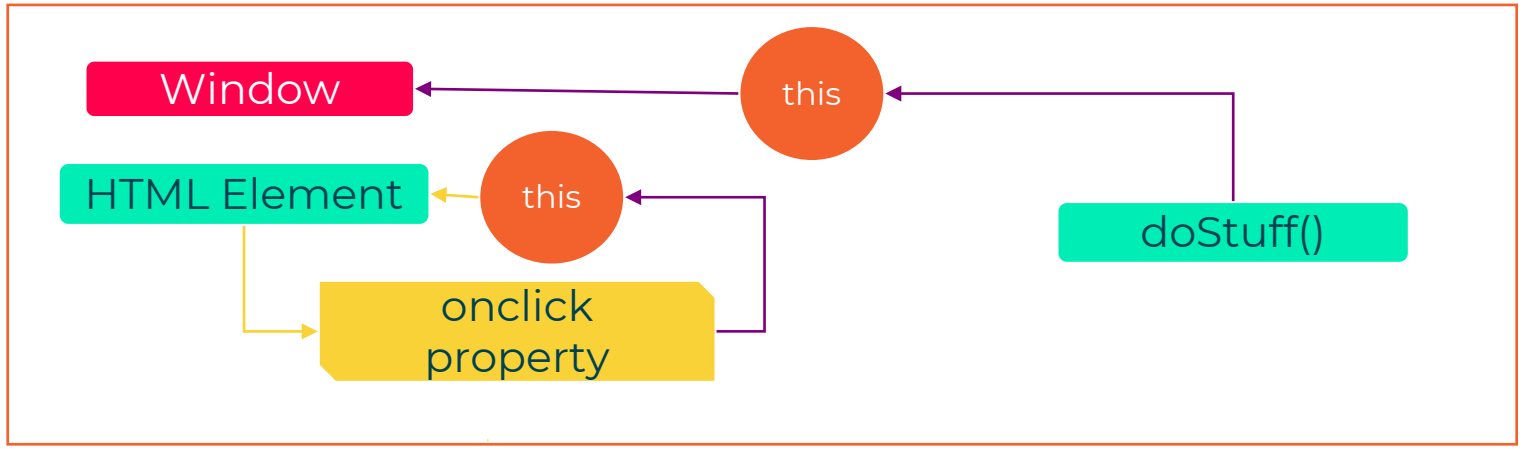
The this keyword

`this` is one of the most useful and powerful keywords in JavaScript.

- Also one of the trickiest to master
- You will see it a lot in the next few modules!

`this` refers to the context within which it is used.

- Every object in JavaScript is referenced to another
- Starting from the window object



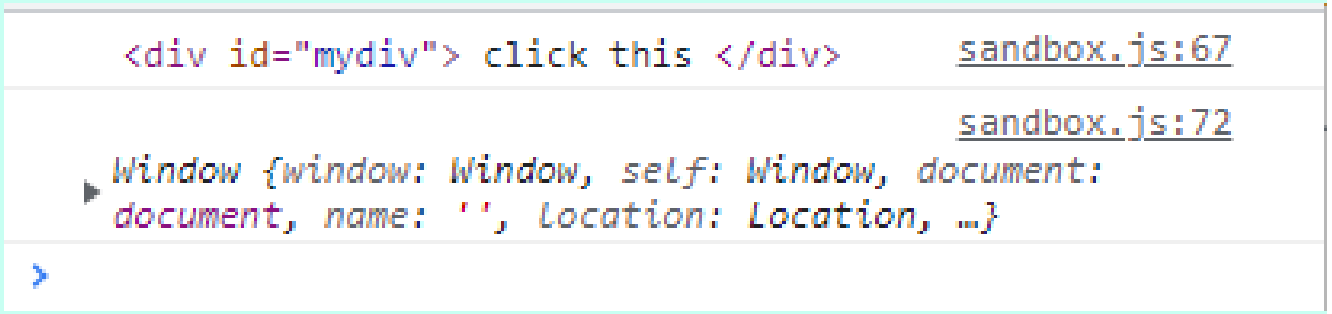
Arrow functions

- Functions create a new context for **`this`** based on how it is called – i.e., if the function belongs to a DOM object and that function is called from the DOM object then **`this`** refers to the DOM object
- Arrow functions are a convenient shorthand for writing an anonymous function that does not create a new context for **`this`**
  - Can be very helpful, but also a gotcha when used in situations, such as event handlers

```
let div = document.querySelector('div');

div.addEventListener('click',function(){
    console.log(this); //DOM Element
});

div.addEventListener('click',()=>{
    console.log(this); //Window
});
```



Arrow functions

- In many situations, we don't need to use **`this`** and so we can use **arrow functions** as a concise alternative
- In some situations, it helps that they do not create a new context

```
button.addEventListener('click', function() {
    this.disabled = true;
    setTimeout(() => {
        alert("Time's up");
        this.disabled = false;
    }, 1000);
});
```

```
                                                    sandbox.js:80
<button id="mybutton">Click me</button>
```

- In this example, the inner arrow function maintains its context to the button, where as if using an anonymous function **`this`** would refer to **`window`** as that is the context when the Timeout expires. Hence, we can re-enable the button from within the timer. Try it both ways and see

Arrow functions Vs Anonymous functions

```
button = document.getElementById('mybutton')
button.addEventListener('click', function() {
    this.disabled = true;
    setTimeout(function() {
        this.disabled = false;
        console.log(this)
    }, 1000);
});
```

sandbox.js:80

```
▶ Window {window: Window, self: Window, document:
  document, name: '', location: Location, …}
```

- In this example, the inner arrow function maintains its context to the button, where as if using an anonymous function **`this`** would refer to **`window`** as that is the context when the Timeout expires. Hence, we can re-enable the button from within the timer. Try it both ways and see

QuickLab 11 - Events

- Adding and removing event handlers from elements

REVIEW

- Understanding JavaScript events
- Subscription models
  - Inline
  - Programmatic
  - Event listeners
- Event bubbling and capturing
- The Event object
- The ‘this’ keyword