Events

PROGRAMMING WITH JAVASCRIPT

QA

Introduction

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

2

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
- Event can be tricky, older browsers implement them badly
    - We will learn how to deal with IE legacies later in the course
- 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

3

Without events there are no scripts. Take a look at any web page with JavaScript in it: in nearly all cases there will be an event that triggers the script. The reason is very simple. JavaScript is meant to add inter*activity* to your pages: the user does something and the page reacts.

Therefore JavaScript needs a way of detecting user actions, so that it knows when to react. It also needs to know which functions to execute, functions that do something that you, the web developer, have judged likely to increase the appeal of your pages. These pages describe the best way to write such scripts. It isn't easy, but it is very satisfying work.

When the user does something an *event* takes place. There are also some events that aren't directly caused by the user: the load event that fires when a page has been loaded, for instance.

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

| 1 User interacts with page | 2 Event is raised | 3 Subscriber function reacts |
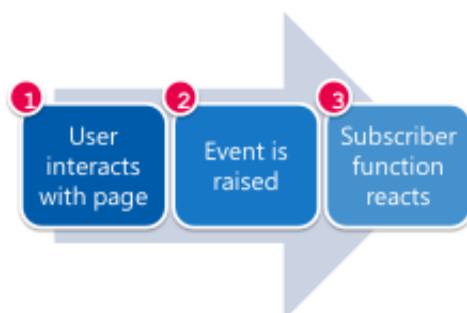
4

The JavaScript event model has three stages. The user (or browser event) occurs. This could be through a mouse clicking a button or a key being pressed. Every DOM object has a list of events it can execute (we will investigate this shortly).

An event may be subscribed to by associating the event to a function. When the event is raised the subscribed function executes. We will examine a number of options for setting up events in the next few slides, the first two only allow a single subscriber function. The third allows multiple subscribers.

The inline subscription model

- The inline subscription model hardcodes events in the HTML
  - It is quick and easy and works in all browser
- 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

The oldest browsers support only one way of registering event handlers, the way invented by Netscape, works in all JavaScript browsers. In the *inline event registration model*, event handlers are added as attributes to the HTML elements they were working on.

Don't use inline registration

Although the inline event registration model is ancient and reliable, it has one serious drawback. It requires you to write JavaScript behaviour code in your XHTML structure layer, where it doesn't belong. In the world of touch, desktop and mobile devices the approach is not sufficient.

Default Actions

Consider an <a> element when the user clicks on a link the browser loads the page specified in its href attribute. This is the *default action* caused by a click event on a link. But what happens when you've also defined an onclick event handler? It should be executed. If the onclick event handler is to be executed at all, it must be done *before* the default action. This has become an important principle of event handling. If an event causes both a default action and execution of a event handling script the event handler script is executed first

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

For simple function calls the simple event registration model is very easy and elegant to use. We already know that HTML elements become DOMElements and that they in turn are objects. Through that object, a series of null value properties are initialised by the browser. As you would of seen in the exercise you just completed, an inline event registration actually creates a function behind the scenes for us. By understanding this, we can leverage the functionality to improve our web development and remove functionality from the markup.

There are two common approaches to using this methodology. The first is to point (reference) the DOM object's event to separate function. This approach should be used when we want to use the same function for multiple elements. It allows code reusability, but means the function could be called by anyone, or used separate from event raising functionality. Obviously, this would cause catastrophic errors.

The alternative approach is to use anonymous functions. When an anonymous function is created, the only reference to the function object is held by the event itself. This creates a 1 to 1 relationship between publisher and subscriber.

One of the key issues with this approach is that only one function can be associated to one event. There is a small work around through creating an anonymous function that calls other functions in a nested hierarchy. This approach is an all or nothing approach and we always have to have the same functionality.

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
        - The event
        - The function to be raised
        - Whether event capturing should occur (optional, default: false)

W3C's DOM Level 2 Event specification pays careful attention to the problems of the traditional model. It offers a simple way to register as many event handlers as you like for the same event on one element.

The W3C event registration model uses the method a**ddEventListener().** You give it three arguments:

- the event type
- the function to be executed
- a Boolean (true or false)

The third Boolean parameter is the trickiest to understand. It states whether the event handler should be executed in the capturing or in the bubbling phase. If you're not certain whether you want capturing or bubbling, use false (bubbling).

We will explore event bubbling in more depth shortly.

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);
```

8

One of the best features of this event registration approach is that we can easily subscribe multiple functions to the same event. In the above example, we have associated the functions callMe and alsoCallMe to the same DOM object. When we click on the object associated to this event, we will fire off a call to both functions. Please note that the W3C model does not state which event handler is fired first and that can not be assumed.

Subscribed functions can be removed from an event at any time using the removeEventListener function:

```
element.removeEventListener('event',funcName,false)
```

This level of granularity allows us to choose which method will be unsubscribed.

If we were to examine the onclick property of our DOM element, we would find that there is no function assigned to it. This is because the event listener approach follows an observer or delegate pattern. Its role is to hold references to the interested functions of that event.

Chrome provides two views to this. In the Elements panel, it shows the DOM elements that raise specific events (note the browser may attach listeners to events for its internal functionality). The Sources panel provides the ability to attach to mouse, keyboard or other types of listeners and when they fire the debugger will automatically capture the event.

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'); });
```

- 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

10

As we discussed earlier, in this chapter it is not at all uncommon that we want to conceal event raising functions, so only the event handler can raise them. The event listener approach also allows us to continue this approach.

This can be extremely useful when we need to pass parameters to the listener. Other than the event itself, no parameter is passed when the event is raised (although, it does maintain scope) instead, we can wrap the functions we want to call inside an anonymous function block and pass the parameter from the DOM element that raises the event.

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

```
<div
id="Element1">

    <button
    id="Element2">
```
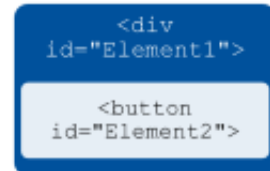
11

The basic problem is very simple. Suppose you have a element inside an element.

Both have an onClick event handler. If the user clicks on element2, they cause a click event in both element1 and element2. But which event fires first? Which event handler should be executed first? What, in other words, is the event order?

Web programming provides two models that stem back to the browser wars of the late 1990's. Netscape said that the event on element1 take `<div id="Element1">` pturing. Microsoft maintained that the event on element2 takes precede

The two event orders are radically op `<button id="Element2">` vides the ability to do both. IE8 and below, followed their approach and though, I caused incredible difficulty to web developers. IE9 onwards follc will cease to be a concern in the future.
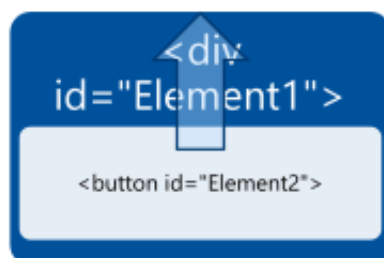
Event capturing

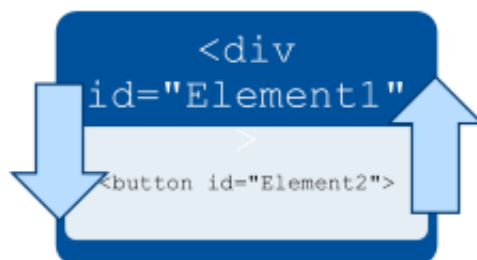- With Event Capturing the event handler of Element1 fires first

Event bubbling

- With event bubbling the event handler of Element2 fires first

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

```
<div
id="Element1"
>
<button id="Element2">
```

14

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

15

The event object also exposes a series of methods:

event.initEvent

Initialises the value of an Event created through the DocumentEvent interface.

event.preventDefault

Cancels the event (if it is cancellable).
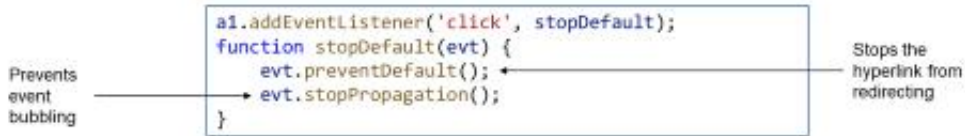
event.stopImmediatePropagation

For this particular event, no other listener will be called – neither those attached on the same element, nor those attached on elements that will be traversed later (in capture phase, for instance).

event.stopPropagation

Stops the propagation of events further along in the DOM.

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);
}
```
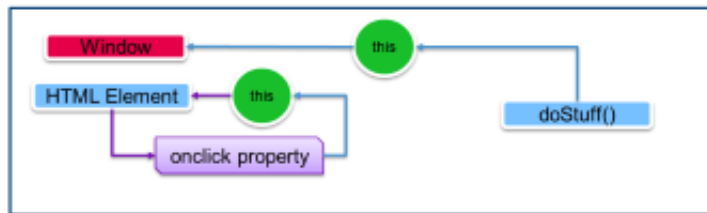
Event handlers may be attached to various objects, including DOM elements, document, the window object, etc. When an event occurs, an event object is created and passed sequentially to the event listeners.

The DOM Event interface is accessible from within the handler function, via the event object passed as the first argument. The following simple example shows how an event object is passed to the event handler function, and can be used from within one such function.

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

In JavaScript, the **this** keyword usually refers to the "owner" of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don't know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation, the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to 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. Which can be very helpful, but also a gotcha when used in situations, such as event handlers

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

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

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

18

In JavaScript, the **this** keyword usually refers to the "owner" of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don't know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to the **window** object.

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);
});
```

- In this example, the inner arrow function maintains its context to the button, where as if using an anonymous function it 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

19

In JavaScript, the **this** keyword usually refers to the "owner" of a function. In the case of event handlers, it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.

Unfortunately, the **this** keyword, though very powerful, is hard to use if you don't know exactly how it works

The question that we'll discuss for the remainder of the page is: What does **this** refer to in the function **doStuff()**? In a page, its owner is the window object.

If we execute **doStuff()** without any more preparation, the **this** keyword refers to the **window** and the function tries to change the **style.color** of the **window**. Since the **window** is a **BOM** and not a **DOM** object, it does not have a **style** object and this fails, with errors.

There is an important difference between inline and programmatic event registration. Inline references the function object, (so there is only ever 1), programmatic event association copies the function object. If we use programmatic event registration, **this** becomes a reference to the calling DOM object. If we use inline registration, **this** refers to the **window** object.

20

Review

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

20

Exercise – programmatic event registration

- Turning the image of the day display into an image gallery
  - Adding event handlers