JavaScript Events

Advanced JavaScript for Web Sites and Web Applications

Events

An event-driven language

- JavaScript is largely an *event driven* language
- When certain things occur within the browser, *Events* are fired.
  - mousedown, mouseup, click, dblclick, keypress, keyup, keydown, mousemove, mouseover, mouseout, focus, load, unload, etc.
- We can write code to run when these events are fired
- These pieces of code are known as *Event handlers*

Event Handlers

- Event handlers:
  - are associated with a specific *event*
  - are attached to specific *elements* in the DOM.
- Many *handlers* can be attached to a specific event/element

Events - addEventListener

- Use **addEventListener()** to attach handlers to an element's events:

```
el.addEventListener(event_type, func_to_run);
```

- Where:
  - **el** is the element to attach the listener to
  - **event_type** is the event to listen for
  - **func_to_run** is the function to execute when the event is fired.

Attaching a handler

- To attach a named function (*myFunction*) as an event handler for a DOM node:

```
el.addEventListener(event_type, myFunction);
```

- Or, use an anonymous function to handle the event:

```
el.addEventListener(event_type, function () {
    // Code to run when event fires
});
```

Events - attaching a *named function* handler

```
// The handler function
function listenerTest() {
    console.log('I am handling an Event!');
}

// The element to attach handler to
var myEl = document.getElementById("headline");

// Adding the event handler to myEl...
myEl.addEventListener("click", listenerTest);
```

Events - attaching an *anonymous function* handler

```
// The element to attach handler to
var myEl = document.getElementById("headline");

// Adding the event handler to myEl...
myEl.addEventListener("click", function () {
    console.log('I am handling an Event!');
});
```

Events - the event object

- When a function is called as an event handler, it gets passed an *event object* as its argument
- This object represents the event that has occurred.
- We can use this object to get information about the event

The event Object

```
// The element to attach handler to
var myEl= document.getElementById("headline");

// Adding the event handler
myEl.addEventListener("click", function (event) {
    // Function receives event object
    console.log(event);
});
```

Events - the event object

- The event object will have different properties, depending on the event that has fired:
  - A *click* event object contains the x and y co-ordinates of the *click*
  - A *keyup* event object contains information about the key that was pressed
- There are some properties which are common to all event objects:
  - `target`, `currentTarget`

Events - event.target

- The `target` property of the event object contains a reference to the element/node which triggered the event.
- This is a regular DOM node, so we can use all the usual methods on it
  - `textContent`, `insertAdjacentHTML`, `classlist`, etc.

Events - event.target

```
<p id="headline">This is my headline</p>
```

```
var myEl = document.getElementById("headline");
myEl.addEventListener("click", function (event) {
    // "event.target" is a DOM node...
    var trigger = event.target;
    // ... so DOM methods are available
    console.log(trigger.textContent);
});
```

Events - event.target

- The `event.target` property is useful when we need to pass the trigger element to another function:

```
function changeText(element) {
    element.textContent= 'I am new text!';
}
var myEl= document.getElementById("headline");
myEl.addEventListener("click", function (event) {
    var trigger = event.target;
    // Pass "clicked" element to function
    changeText(trigger);
});
```

Events - default behaviour

- For some events, the browser has built-in functionality that is activated when the event is fired.
  - E.g. When the “click” event fires for an <a> element, the browser opens the page referenced in the `href` attribute
- This default behaviour occurs *after* any event handlers attached to the element have completed.

Events - preventing default behaviour

- If we don't want the default behaviour to occur, we can stop it from inside our handler function.
- We do this with the `preventDefault()` method of the event object:

```
event.preventDefault();
```

- We commonly use `preventDefault()` when dealing with links...

Events - preventing default behaviour

```
<a href="index.html" id="my-link">Link</a>
```

```
var link = document.getElementById("my-link");
link.addEventListener("click", function (event) {
    // Prevent the default behaviour
    event.preventDefault();
    console.log('You clicked... then nothing');
});
```

Event Bubbling

Events - bubbling

- For some event types, when they are triggered:
  - They are fired on the element that triggered them: handlers attached to that element run.
  - Then, the event is fired for each *ancestor* of the trigger element.
  - If any of the ancestor elements have handlers attached to *the same event*, they will also run
- This is called *event propagation* (or *event bubbling*)

Events - bubbling example

```
<div id="content">
    <p id="headline">This is my headline</p>
</div>
```

```
var inner = document.getElementById("headline"),
    outer = document.getElementById("content");
inner.addEventListener("click", function (event) {
    console.log('I am the inner element');
});
outer.addEventListener("click", function (event) {
    console.log('I am the outer element');
});
```

Events - bubbling example

- In the previous example, if the user clicks on the “headline” element:
  - First, the event handler attached to the “headline” element will run
  - Then, the handler attached to the “content” element will run
- Both handlers will receive an event object, but the object will be slightly different

Events - bubbling

- For every handler in the chain, the `event.target` property will *always* be a reference to the element that initially fired the event.
  - In the example, this will be the “headline” element
- But the object also has a `currentTarget` property which contains a reference to the element who’s handler is *currently being executed*

```
event.currentTarget;
```

Events - bubbling

- Event bubbling useful, but we may not always want it to occur.
- The event object has a method we can use to prevent the event from bubbling up through the DOM

```
event.stopPropagation();
```

- Any function in the propagation chain can call this method, and it will cause the event to stop bubbling *at that point* in the DOM

Exercise 1

- Download the *Session 5 exercises* document from Moodle and do *Exercise 1*

Delegation

Event delegation

- *Event delegation* is a pattern which takes advantage of *bubbling* in order to write more efficient and flexible event handlers.

Events - the bubble chain

- What we know:
  - Each handler in the *chain* receives the *event object* as an argument.
  - The *event object* has a `target` property that contains a reference to the *element that triggered the event*.
- In other words, at any point in the *chain*, we can determine which element initially triggered the event

Event delegation

- Therefore, we can:
  - declare a single event handler on an element that has multiple children
  - inside the handler function, use `event.target` to find out which *child* triggered the event.
  - use this information to execute appropriate code

Events delegation example

```
<div id="wrapper">
    <p class="button">One</p>
    <p class="button">Two</p>
</div>
```

```
var w = document.getElementById('wrapper');
w.addEventListener('click', function (event) {
    // Get textContent of clicked element
    var cText = event.target.textContent;
    console.log(cText);
});
```

Exercise 2

- Now do *Exercise 2* from the Session 5 exercises

Event delegation problems

- There is a small problem with our delegation example.
- If you click inside the `button-wrapper` element, but not on an actual button, the event is still fired!
- We need to perform some checks within our handler that stop this from happening.

Event delegation problems

- We can do this by examining the `target` element and ensuring it is one we are interested in before continuing.
- We could do one of these:
  - Check if the element is a “p” tag (use `nodeName`)
  - Check if the element has the class *button* (use `classList.contains`)
  - Check if the element has a *data-action* attribute (use `getAttribute`)

Event delegation (fine tuning)

- Task: Implement one of these tests within your event handler from exercise 2
- When choosing which method to use, consider that other elements, which we do not want our code to attach to, may get added to the wrapper

Why is delegation useful?

- We can write less code and declare fewer event handlers
- If we add elements to the DOM dynamically, we do not have to attach new event handlers to them.
  - If the event handler was attached to `.button`, a dynamically added `.button` element would not have the event listener attached to it automatically

Exercise 3

- Now do *Exercise 3* from the Session 5 exercises

Custom Events

Custom Events

- We can also create our own events which other modules/scripts can listen for.
- The steps involved:
  - Create an event object
  - Other code will attach handlers/listeners to your event
  - Within your code/module, fire the event

A basic custom event

```
// Create event (within module or elsewhere)
var myEvent = new Event('HiMom');

// Other code listens for "HiMom" event
document.addEventListener('HiMom',
    function(event) {
        console.log('Custom event has fired');
    }
);

// When appropriate, dispatch the event.
document.dispatchEvent(myEvent);
```

Custom events - Elements

- In the previous example, the *event* was dispatched on the *document* element.
- Also, the *listener* was added to the *document* element
- However, you can dispatch the event on any element you like
  - as long as the listeners are added to the same element, they will work

Custom events - passing data

- Quite often, we want to pass data with our event.
- To do this, we use the `CustomEvent` constructor.
- Using this constructor, we can pass an object with a `detail` property, which is itself, another object
  - The `detail` object contains property/value pairs we want to pass to the event handlers

CustomEvent constructor - passing data

```
// The options object...
// "detail" is where our data goes
var opts = {
    "detail": {
        "message": "I am data"
        "value": 7
    }
};
// Pass options object to constructor
var myEvent = new CustomEvent('HiMom', opts);
```

Custom events - receiving data

- When event handlers are invoked by our event being fired, they will receive the `detail` object as a property of the regular *event object*
- Standard *dot notation* is used to access the data contained in it:

```
event.detail.message;
event.detail.value;
```

Custom events - receiving data

```
// Accessing the "detail" object
// via the "event" object
document.addEventListener(
    'HiMom',
    function (event) {
        var message = event.detail.message;
        var value = event.detail.value;
        console.log(message + ' '+ value);
    }
);
```

Custom events - options

- Using the *options* argument of the *CustomEvent* constructor, we can also specify:
  - whether or not the event should *bubble*
  - whether event handlers should be able to prevent the event's default behaviour (with `preventDefault()`)

Custom events - options

```
// Setting other options for event
opts = {
    "bubbles": true,
    "cancelable": false,
    "detail": {
        "message": "I am data",
        "value": 7
    }
};
var myEvent = new CustomEvent('HiMom', opts);
```

Custom events - with modules

- When used with modules, custom events help us to achieve loose coupling between the components of our applications.
  - Loose coupling: functions/modules do not depend upon the presence of other functions/modules in order to do their job.

Custom events - with modules

- Open `vat-calculator-before.js` from the workshop 5 download pack
- This is a simple *revealing module* which handles tax calculations

Custom events - with modules

- Now imagine another part of our app that is responsible for displaying the current VAT rate on the page.
- This function/module is separate to the `VatCalculator` module:

```
function updateVatDisplay(value) {
    var el = document.getElementById('vat-total');
    el.textContent = 'VAT rate is: ' + value;
}
```

Custom events - with modules

- When the page loads, we can retrieve the VAT rate from the vat calculator and pass it to this function:

```
var vat = VatCalculator.get();
updateVatDisplay(vat);
```

- Result: the page accurately displays the current VAT rate (as stored in the `VatCalculator` object)

Custom events - with modules

- But what happens when code elsewhere in our app calls the `increase` method of the VatCalculator?
  - i.e. what happens when the VAT rate changes?
- Ideally, we want to call `updateVatDisplay()` again, passing it the new rate.
  - otherwise the value displayed on the page will be wrong!

How can we do this?

- We could call `updateVatDisplay()` from within the `raiseVat()` method of the `VatCalculator` module:

```
raiseVat = function (amount) {
    rate = rate + amount;
    updateVatDisplay(rate);
}
```

- But now `VatCalculator` is dependent upon the `updateVatDisplay` function
  - i.e. They are *Tightly Coupled*!

Custom events - with modules

- One solution to this problem is to use custom events.
  - i.e. We fire a custom event every time the vat rate changes
- Code outside the module can listen for our event and react accordingly.

Step 1: Create the event object:

```
var VatCalculator = (function () {
    var rate = 20, vatEvent, raiseVat, etc;
    // Create the event object, adding
    // the "rate" to the "detail" object
    vatEvent = new CustomEvent("VatRaised", {
            "detail": {
                "rate": rate
            }
        }
    );
    // Other methods omitted for clarity...
})();
```

Step 2: Firing the event

```
var VatCalculator = (function() {
    // Other stuff omitted for clarity..
    raiseVat = function(amount) {
        rate = rate + amount;
        // Rate has changed:
        // so update value in event object
        vatEvent.detail.rate = rate;
        // Now fire the event...
        document.dispatchEvent(vatEvent);
    };
    // Other stuff omitted for clarity...
})();
```

Step 3: Code outside the module listens for event

```
// Event is dispatched from "document"
// so we add handler to "document"
document.addEventListener(
    "VatRaised",
    function (event) {
        // get the new rate from detail object
        var newVat = event.detail.rate;
        // Pass rate to function
        updateVatDisplay(newVat);
    }
);
```

All steps combined

- See `vat-calculator.js` in the workshop files for the complete code.
- Open `vat-calculator/test.html` in your browser and watch the console to see it in action.

Custom events - Why?

- When we use events in this way, our modules immediately become more flexible and re-usable.
  - Any other modules or components we use in our application can *listen* for events fired from our modules.
  - We avoid *tight coupling*, making it easier to copy components between different applications

Custom events

- A good example of custom events can be seen with the Youtube player API, which fires events when:
  - The player and video have loaded
  - The user clicks “play”, “pause”, etc.
- By listening for these events, you can update other elements when the user interacts with the video
  - E.g. stop things that might interfere with the video

Exercise 4

- Now do *Exercise 4* from the Session 5 exercises