Lesson10: Event Handling

SPONTANEOUS Right Action

Maharishi University of Management -Fairfield, Iowa © 2019

Main Point Preview

Event handlers take callback functions that are executed later when the event occurs.

Science of Consciousness: Callbacks are a form of memory for an action that is automatically executed when an event happens. When we act from deep levels of awareness we are more likely to activate appropriate memories and reactions (event handlers).

Mouse Events

`click`	user presses/releases mouse button on the element
`dblclick`	user presses/releases mouse button twice on the element
`mousedown`	user presses down mouse button on the element
`mouseup`	user releases mouse button on the element movement
`mouseover`	mouse cursor enters the element's box
`mouseout`	mouse cursor exits the element's box*
`mousemove`	mouse cursor moves around within the element's box

* or exits any descendent. jQuery has mouseleave which only fires for the element, not descendants

Page/Window Events

- **window events**
  - `load`, `unload` the browser loads/exits the page
  - `resize` the browser window is resized
  - `error` an error occurs when loading a document or an image
  - `contextmenu` the user right-clicks to pop up a context menu

- **Form events**
  - `submit` form is being submitted
  - `reset` form is being reset
  - `change` the text or state of a form control has changed

Keyboard/text events

- `keydown` user presses a key while this element has keyboard focus
- `keyup` user releases a key while this element has keyboard focus
- `keypress` user presses and releases a key while this element has keyboard focus
- `focus` this element gains keyboard focus
- `blur` this element loses keyboard focus
- `select` this element's text is selected or deselected)

- **Keyboard events object properties**
  - `which` ASCII integer value of key that was pressed (convert to char with String.fromCharCode)
  - `altKey`, `ctrlKey`, `shiftKey` true if Alt/Ctrl/Shift key is being held

Recall `window.onload` event

- We want to attach our event handlers right after the page is done loading (Why?)
  - There is a global **event** called `window.onload` event that occurs at that moment

```
// this will run once the page has finished loading
function functionName() {
   element.event = functionName;
   element.event = functionName;
   ...
}

window.onload = functionName; // DOM version
$(document).on("ready", functionName); //jQuery version (runs sooner than
   onload)
```

Attaching event handlers the jQuery way

- To use jQuery's event features, you must pass the handler using the jQuery object's event method

```
DOMObject.onevent = function; //DOM way
jQueryObject.event(function); //jQuery way
jQueryObject.on("event", function); //jQuery way

// call the playNewGame function when the Play button is
  clicked
$("#play").click(playNewGame);

function playNewGame(evt) {
 // respond to the click event
}
```

You can trigger the event manually by calling the same function with no parameters

```
$("#play").click();
```

The jQuery event object

- Event handlers can accept an optional parameter to represent the event that is occurring. Event objects have the following properties and methods:

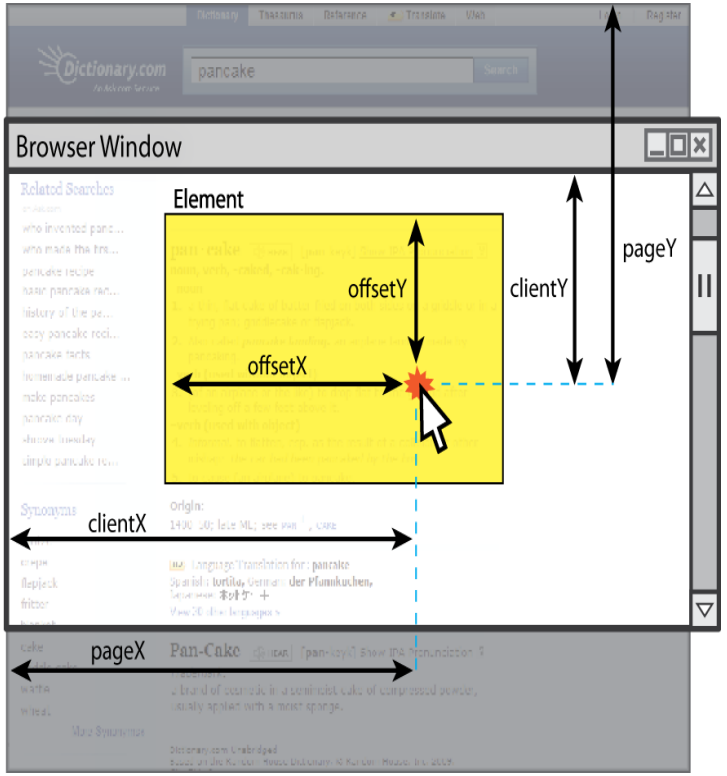
```
function handler(event) {
 // an event handler function ...
}
```

target	The element on which the event handler was registered
preventDefault()	Prevents browser from performing its usual action in response to the event
stopPropagation()	Prevents the event from bubbling up further
stopImmediatePropagation()	Prevents the event from bubbling and prevents any other handlers from being executed

Mouse Event Object

- The event object is passed to a mouse handler has these properties:

`clientX`, `clientY`	coordinates in browser window
`screenX`, `screenY`	coordinates in screen
`offsetX`, `offsetY`	coordinates in element (non-standard)
`pageX`, `pageY`	coordinates in entire web page in which mouse button was clicked



Example

```
<pre id="target">Move the mouse over me!</pre>

$(function(){
 $("#target").on('mouseover', showCoords);
});

function showCoords(evt) {
 $("#target").html(
  "page : (" + evt.pageX + ", " + evt.pageY + ")\n" +
  "screen : (" + evt.screenX + ", " + evt.screenY + ")\n"
  +
  "client : (" + evt.clientX + ", " + evt.clientY + ")"
 );
}
```

Main Point

Event handlers take callback functions that are executed later when the event occurs.

Science of Consciousness: Callbacks are a form of memory for an action that is automatically executed when an event happens. When we act from deep levels of awareness we are more likely to activate appropriate memories and reactions (event handlers).

Main Point Preview

JavaScript code runs inside of an object and the 'this' keyword refers to that object. Event handlers that are attached unobtrusively are bound to that element and inside the handler 'this' references the bound DOM element. Usage of 'this' in event handlers is a common JavaScript programming idiom that enables handlers to be reused across different kinds of elements.

Science of Consciousness: We can think of the TM Technique as an event handler that gives the result of transcending and can be used by any person (element).

jQuery and **this**

- Recall
  - All JavaScript code actually runs inside of an object
  - By default, code runs in the global window object
    - (so this === window)
  - All global variables and functions you declare become part of window
- In jQuery you need to understand what the current object will be when the `this` keyword is used

Event handler binding

Event handlers attached unobtrusively are bound to the element. Inside the handler, that element becomes `this` (rather than the window)

```
$(function() {
 $("#textbox").mouseout(sayHi);
 // bound to text box here
 $("#submit").click(sayHi);
 // bound to submit button here
});

function sayHi() {
 // sayHi knows what object it was called on
 this.value = "sayHi";
}

<div class="exampleoutput">
 <input id="textbox" />
 <input type="submit" id="submit" value="Save">
</div>
```

- See example: lecture09_examples/demo2.html

Fixing redundant code with `this`

```
<fieldset>
 <label><input type="radio" name="ducks" value="Huey" /> Huey</label>
 <label><input type="radio" name="ducks" value="Dewey" /> Dewey</label>
 <label><input type="radio" name="ducks" value="Louie" /> Louie</label>
</fieldset>

$(":radio").click(processDucks);

function processDucks() {
 if ($("huey").checked) {
   alert("Huey is checked!");
 } else if ($("dewey").checked) {
   alert("Dewey is checked!");
 } else {
   alert("Louie is checked!");
 }
 alert(this.value + " is checked!");
}
```

If the same function is assigned to multiple elements, each gets its own bound copy

Main Point

JavaScript code runs inside of an object and the 'this' keyword refers to that object. Event handlers that are attached unobtrusively are bound to that element and inside the handler 'this' references the bound DOM element. Usage of 'this' in event handlers is a common JavaScript programming idiom that enables handlers to be reused across different kinds of elements.

Science of Consciousness: We can think of the TM Technique as an event handler that gives the result of transcending and can be used by any person (element).

Main Point Preview

Events bubble from the bottom of the DOM tree to the top. The jQuery stopPropagation method prevents bubbling up the element tree. jQuery's stopImmediatePropagation method prevents any other handlers that might be attached to the current element from being executed.

Science of Consciousness: In the DOM, events can affect almost every element. In the world almost everything is connected, and it is impossible to intellectually predict all the ramifications of an action. If our thoughts are connected to the home of all the laws of nature, then our actions will spontaneously be in accord with the entire environment.

Stopping an event's browser behavior

- To abort a form submit or another event's default browser behavior, call jQuery's **preventDefault()** method on the event

```
<form id="exampleform" action="">...</form>


$(function() {
 $("#exampleform").on('submit', checkData);
});


function checkData(event) {
 if ($("#firstname").val() == "" || $("#lastname").val().length
  != 2) {
        alert("Error, invalid firstname/lastname.");
        event.preventDefault();
 }
}
```

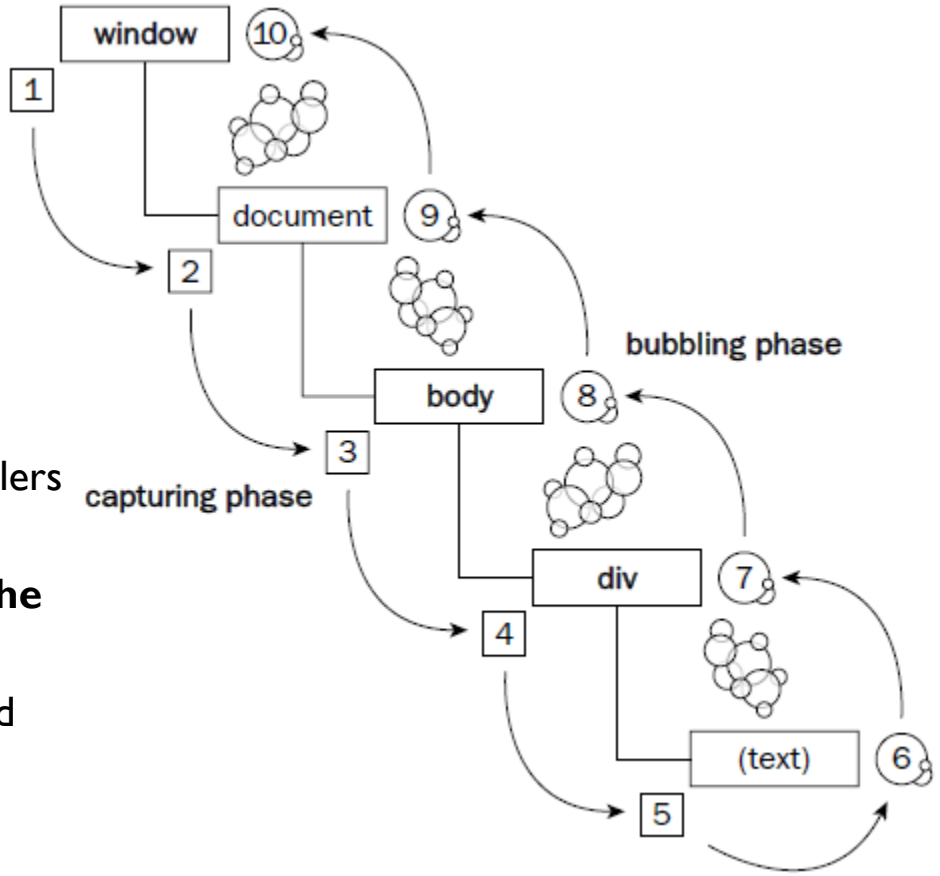
Which element gets the event?

```
<body>
 <div>
   <p> Events are <em>crazy</em>!</p>
 </div>
</body>

$(function() {
 $("body, div, p, em").click(hello);
});

function hello() {
 alert("You clicked on the " + $(this)[0].tagName);
}
```

- What happens when I click on the em? Which element will get the event?
- Answer: All of them!

Event Bubbling

```
<body>
 <div>
   <p> Events are <em>crazy</em>!
   </p>
 </div>
</body>
```



- Clicking the `em` is actually a click on every element in this page.
- Therefore it was decided that all of the handlers should be executed.
- The events **bubble from the bottom of the DOM tree to the top.**
- The opposite model (top to bottom) is called **capturing and is not widely used.**

Stopping an event from bubbling

- Use the **stopPropagation()** method of the jQuery event to stop it form bubbling up.

```
<body>
 <div>
  <p> Events are <em>crazy</em>! </p>
 </div>
</body>

$(function() {
 $("body, div, p, em").click(hello);
});

function hello(evt) {
 alert("You clicked on the " + this.nodeName);
 evt.stopPropagation();
}
```

Run Example! Or demo: lecture09_examples/demo4.html

Multiple handlers

```
<body>
 <div>
    <p> Events are <em>crazy</em>! </p>
 </div>
 <p>Another paragraph!</p>
</body>

$(function() {
 $("body, div, p, em").click(hello);
 $("div > p").click(anotherHandler);
});

function hello() {
 alert("You clicked on the " + this.nodeName);
}

function anotherHandler() {
 alert("You clicked on the inner P tag");
}
```

What happens when the first p tag is clicked? Run Example or demo: lecture09_examples/demo5.html

Stopping an event right now

- Use **stopImmediatePropagation()** to prevent any further handlers from being executed.
- Handlers of the same kind on the same element are otherwise executed in the order in which they were bound.

```
function anotherHandler(evt) {
 alert("You clicked on the inner P
  tag");
 evt.stopImmediatePropagation();
}
```

Run Example!

stopImmediatePropogation()

```
$("div a").click(function() {
 // Do something
});

$("div a").click(function(evt) {
 // Do something else
 evt.stopImmediatePropagation();
});

$("div a").click(function() {
 // THIS NEVER FIRES
});

$("div a").click(function() {
 // THIS NEVER FIRES
});
```

Only the first two handlers will ever run when the anchor tag is clicked.

jQuery handler return value

- jQuery does something special if you return false in your event handler
  1. prevents the default browser action, eg evt.preventDefault()
  2. stops the event from bubbling, eg evt.stopPropagation()

```
<form id="exampleform"> ... <button>Done</button> </form>

$(function() {
 $("#exampleform").submit(cleanUpData);
 $("button").click(checkData);
});

function checkData() {
 if ($("#city").val() == "" || $("#state").val().length != 2) {
   alert("Error, invalid city/state."); // show error message
   return false;
 }
}
```

Event delegation for DOM events

- Very useful with collections or lists with elements that all have events
  - E.g., list items in mobile apps

Algorithm:

Put a single handler on the container.

In the handler – check the source element event.target.

If the event happened inside an element that interests us, then handle the event.

Benefits:

Simplifies initialization and saves memory: no need to add many handlers.

Less code: when adding or removing elements, no need to add/remove handlers.

Limitations:

event must be bubbling.

Some events do not bubble. (e.g., blur, focus, load, mouseenter, …)

low-level handlers should not use event.stopPropagation().

Main Point

Events bubble from the bottom of the DOM tree to the top. The jQuery stopPropagation method prevents bubbling up the element tree. jQuery's stopImmediatePropagation method prevents any other handlers that might be attached to the current element from being executed.

Science of Consciousness: In the DOM, events can affect almost every element. In the world almost everything is connected, and it is impossible to intellectually predict all the ramifications of an action. If our thoughts are connected to the home of all the laws of nature, then our actions will spontaneously be in accord with the entire environment.

Main Point Preview

JavaScript is single threaded. It handles asynchronous events by storing them and cycling through them in an 'event loop'.

Science of Consciousness: The event loop gives the appearance of multitasking even though there is only ever a single task and thread of execution. The universe appears to be infinitely diverse even though there is only a single unified field.

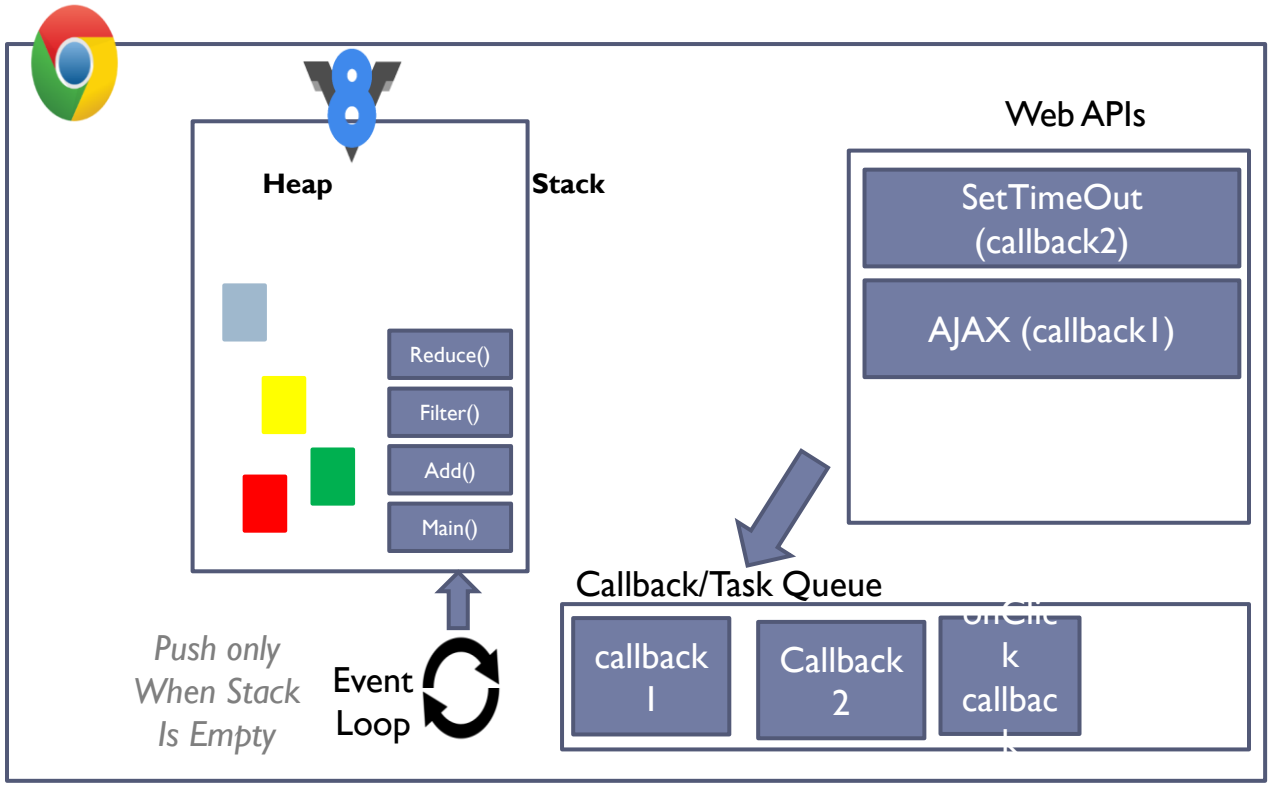
Asynchronous & Callbacks

A callback function is a function you give to another function, to be invoked later by the other function. Callbacks can be synchronous or asynchronous, as in the case of event handlers and timers.

Since JS in the browser is single threaded, how can we handle asynchronous callbacks?

Asynchronous callback functions are handled in JavaScript by an **Event/Task Queue**.

Concurrency & the Event Loop



One thing at a time? Not really!

If you block the stack, browser can't do the render queue

JS Timers (review)

```
setTimeout(function, delayMS); // arranges to call given function after given delay in ms
setInterval(function, delayMS); // arranges to call function repeatedly every delayMS ms
```

Both **setTimeout** and **setInterval** return an ID representing the timer, this ID can be passed to **clearTimeout(timerID)** and **clearInterval(timerID)** to stop the given timer.

Note: If **function** has parameters: **setTimeout(function, delayMS, param1, param2 ..etc);**

```
setTimeout(hideBanner, 5000);
function hideBanner() { // called when the timer goes off
  document.getElementById("banner").style.display = "none";
}
```

Alarm clock example. 

Callbacks and Events Queue

```
// In what order will the results be printed and why?

console.log(1);
var a = setTimeout(function(){ console.log(2); }, 1000);
var b = setTimeout(function(){ console.log(3); }, 0);
console.log(4);
```

hint: when an event ‘fires’ the handler is put into the event queue. It is called when the call stack is empty and the event loop takes the next task from the event queue

Main Point

JavaScript is single threaded. It handles asynchronous events by storing them and cycling through them in an 'event loop'.

Science of Consciousness: The event loop gives the appearance of multitasking even though there is only ever a single task and thread of execution. The universe appears to be infinitely diverse even though there is only a single unified field.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Spontaneous Right Action

1. Event handling is a fundamental aspect of JavaScript programming. jQuery makes it easy to attach event handlers to DOM elements.

2. Some subtle aspects of JavaScript event handlers include the use of event arguments passed to event handlers depending on the type of element, the use of the keyword 'this' that can refer to different objects since functions are first class in JavaScript, and the need to sometimes control event propagation.

---

3. **Transcendental consciousness**. The home of all the laws of nature

4. Impulses within the transcendental field: Thoughts arising from this level will be able to spontaneously respond with right actions to events because they are supported by all the laws of nature.

5. Wholeness moving within itself: The unified field is a singularity that appears as diversity