JavaScript in the Web Browser

Lecture 4 – CPEN400A

Based on the book “JavaScript: The Definitive Guide”, David Flanagan, O’Reilly

Outline

- **Browsers and web application model**
- The Window object: setTimeOut, alert etc.
- Event Handling in Modern Browsers
- Event Propagation in the DOM

Browser as an OS !

- Modern Browsers are equivalent to an OS for web applications
  - Provide core services such as access to the display (DOM, location bar), and permanent state (cookies, local storage, history)
  - Schedule event handlers for different tasks and control the global ordering of events
  - Allow network messages to be sent and received from the server

Modern Web Application

- Applications running on web browsers that use the browser's facilities
  - Update the browser's DOM or shared location bar
  - Schedule events in the future and register event handlers for various parts of the web application
  - Send and receive asynchronous AJAX messages from the web server

- Web applications run on top of the browser OS !

Browser Sandbox

- However, web applications are restricted in their behaviour for security reasons
  - Cannot write persistent state to the host file system (use cookies or browser local storage)
  - Cannot write to parts of the DOM tree that come from other domains (Same Origin Policy - SOP)
  - Cannot read cookies belonging to other domains (SOP)
  - Only allowed to communicate with their domain

Same Origin Policy (SOP)

- Restricts which parts of the web application can be read/written by JavaScript code
- Origin = (URL, domain, portNumber)
- NOTE: Origin of the script is not important. What is important is the origin on the document in which script is embedded

Four ways to include JS code in a Web Application: Method 1

- Directly using <script> and </script> tag
  - Easy method; fast for the browser to download
  - Suitable for small 'utility' scripts on the page
  - Is treated as though it's in the same domain

```
<script>
// JavaScript code goes here
</script>
```

Four ways to include JS code in a Web Application: Method 2

- Using <script src=“”> and </script> tag
  - Recommended method
  - Nothing should appear between script and /script
  - Allows sharing and caching of JavaScript files
  - It’s treated as though it in the same domain

<script src=“somefile.js”>

</script>

Four ways to include JS code in a Web Application: Method 3

- Within event handlers in the webpage
  - To trigger code when the event occurs
  - Typically name of JS function to call to handle the event, but can be any arbitrary piece of code
  - This is deprecated now (DOM level 0)

```
<input type="checkbox" name="options"
        onchange="checkBoxChanged">
```

Four ways to include JS code in a Web Application: Method 4

- Directly in the URL tage preceded by javascript:
  - Not recommended anymore
  - Different browsers treat return values differently
  - Used to support bookmarklets (more later)

<a href="javascript:alert('Clicked');">

JavaScript Execution Model

- Browser follows two phase execution model

- Phase 1:
  - All code within the <script></script> tag is executed when they're loaded in the order of loading (unless the script tag is async or deferred)
  - Some scripts may choose to defer execution or execute asynchronously. These are executed at the end of phase 1

JavaScript Execution Model (contd)

- Phase 2:
  - Waits for events to be triggerred and executes handlers corresponding to the events in order of event execution (single-threaded model)
  - Events can be of four kinds:
    - Load event: After page has finished loading (phase 1)
    - User events: Mouse clicks, mouse moves, form entry
    - Timer events: Timeouts, Interval
    - Networking: Async messages response arrives

Outline

- Browsers and web application model
- **The Window object: setTimeOut, alert etc.**
- Event Handling in Modern Browsers
- Event Propagation in the DOM

Window object

- Global object that provides a gateway for almost all features of the web application
- Passed to standalone JS functions, and can be accessed by any function within the webpage
- Example Features
  - DOM: Through the window.document property
  - URL bar: Through window.location property
  - Navigator: Browser features, user agent etc.

window.alert, confirm and prompt

- Alert: Simple way to popup a dialog box on the current window with an OK button
  - Can display an arbitrary string as message
- Prompt: Asks the user to enter a string and returns it
- Confirm: Displays a message and waits for user to click OK or Cancel, and returns a boolean

Examples

```
do {
    var name = prompt(“What is your name?”);
    var correct = confirm(“You entered: “ +
name);
} while (! correct);
// This is bad security practice – don’t do this !
alert(“Hello “ + name);
```

setTimeout and setInterval

- setTimeout is used to schedule a future event asynchronously **once** after a specified no of milliseconds (can be set to 0)
  - Can specify arguments to event handler
  - Can be cancelled using the clearTimeout method

- *setInterval* has the same functionality as *setTimeout*, except that the event fires repeatedly until *clearInterva*l is invoked

Example of setTimeout

```
var timeoutHandler = function(message) {
    return function() {
        alert(message);
    };
};

var ret = setTimeout(timeoutHandler("Hello"), 100);
…
if (flag) clearTimeout(ret);
```

Example of setInterval

```
var intervalHandler = function(message) {
    var i = 0;
     return function() {
          alert(message + ‘  ‘ + i);
          i+ = 1;
    }
};

var ret = setInterval(intervalHandler(“invocation”), 1000);
….
if (flag) clearInterval(ret);
```

Class Activity

- Create a new function that invokes another function a specified number of times noTimes, asynchronously, each time after ‘time’ ms. Also, the function should pass on the number of times it calls the invoked function to it as an argument.
- HINT: You can do it through setTimeout or setInterval

```
function invokeTime( func, noTimes, time) {
    ….
}
```

Outline

- Browsers and web application model
- The Window object: setTimeOut, alert etc.
- **Event Handling in Modern Browsers**
- Event Propagation in the DOM

Event Handling

- JavaScript code is event-driven, which means that you need to register event callbacks

- Events are of five types in JavaScript
  - Mouse Events (e.g., mouseclick, mousemove, etc)
  - Window Events (load, DOMContentLoaded, etc)
  - Form events (submit, reset, changed etc)
  - Key events (keydown, keyup, keypress etc)
  - DOM events (part of DOM3 specification)

A cautionary note on event handling

- There are many browser incompatibilities regarding the types of events implemented, and the way to register event handlers (e.g., IE prior to v9 is different from almost all other browsers)
- This is complicated by the fact that the DOM3 spec itself is a moving target for over 10 years
- In this class, we will follow DOM2 spec. and assume that the browser is standard compliant
  - Focus on set of events that are common (except IE)

Registering Event Handlers

- Two ways of registering event handlers
  - Old method (DOM 1.0): Directly add a ‘onclick’ or ‘onload’ property to the DOM object/window
    - Disadvantage: Allows only one event handler to be specified. New handlers must remember to chain the old handler, and can potentially ‘swallow’ the handler
  - New method (DOM 2.0): Allows multiple event handlers to be added to the DOM object/window

Registering Event handlers: DOM 1.0

- Use ‘on<event>’ as the handler for <event>
  - No caps anywhere. Eg., onload, onmousemove

```
element.onclick = function(event) {
      this.style.backgroundcolor = ‘#ffffff’;
     return true;
}
```

1. this is bound to the DOM element on which the onclick handler is defined – can access its properties thro’ *this.prop*

2. return value of *false* tells browser not to perform the default value associated with the property (*true* otherwise)

Chaining event handlers in DOM 1.0 method (This is deprecated now !)

- If you want to have multiple event handlers in the above method, you need to remember to chain the earlier handlers and call them

```
var old = element.onclick;
element.onclick = function(event) {
        this.style.backgroundcolor = '#ffffff';
        if (old) return old(event);
        return true;
    }
```

Registering Event handlers: DOM 2.0

- The DOM 1.0 method is clunky and can be buggy. Also, difficult to remove event handlers

- DOM 2 event handlers
  - addEventListener for adding a event handler
  - removeEventListener for removing event handlers
  - stopPropagation and stopImmediatePropagation for stopping the propagation of an event (later)

addEventListener

- Used to add an Event handler to an element. Does NOT overwrite previous handlers
  - Arg1: Event type for which the handler is active
  - Arg2: Function to be invoked when event occurs
  - Arg3: Whether to invoke in the ‘capture’ phase of event propagation (more later) - false typically

```
var b = document.getElementById(“mybutton”);
b.addEventListener(“click”, function() {
                                        alert(“hello”); },
                            false);
```

More on addEventListener

- Does not overwrite previous handlers, even those set using onClick, onMouseOver etc.
- Can be used to register multiple event handlers – invoked in order of registration (handlers set through DOM 1.0 model have precedence)

```
var b = document.getElementById("mybutton");
b.addEventListener("click", function() {
                                    alert("hello"); },  false);
b.addEventListener("click", function() {
                                    alert("world"); },  false);
```

removeEventListener

- Used to remove the event handler set by addEventListener functions, with the same arguments
  - No error even if the function was not set as event handler

```
var handleMouseclick = function() {
                                    alert(“clicked”);
                                };
var b = document.getElementById(“mybutton”)
b.addEventListener(“click”, handleMouseClick,  false);
b.removeEventListener(“click”, handleMouseClick, false);
```

Event Handler Context

- Invoked in the context of the element in which it is set (*this* is bound to the target)
- Single argument that takes the *event* object as a parameter – different events have different properties, with info about the event itself
- Return value is discarded – not important
- Can access variables in the scope in which it is defined, as any other JS function
  - Can support closures within Event Handlers

Class Activity

- Write a handler for the ‘click’ property of the button in the example earlier that displays a message (str1 + str2) using the alert feature
  - str1 is determined at runtime when setting the event handler for the button ‘b’, and should not be stored in the global context
  - str2 is determined based on the event target at the time of its invocation e.g., event.target. This may be different from the button ‘b’ (later why).

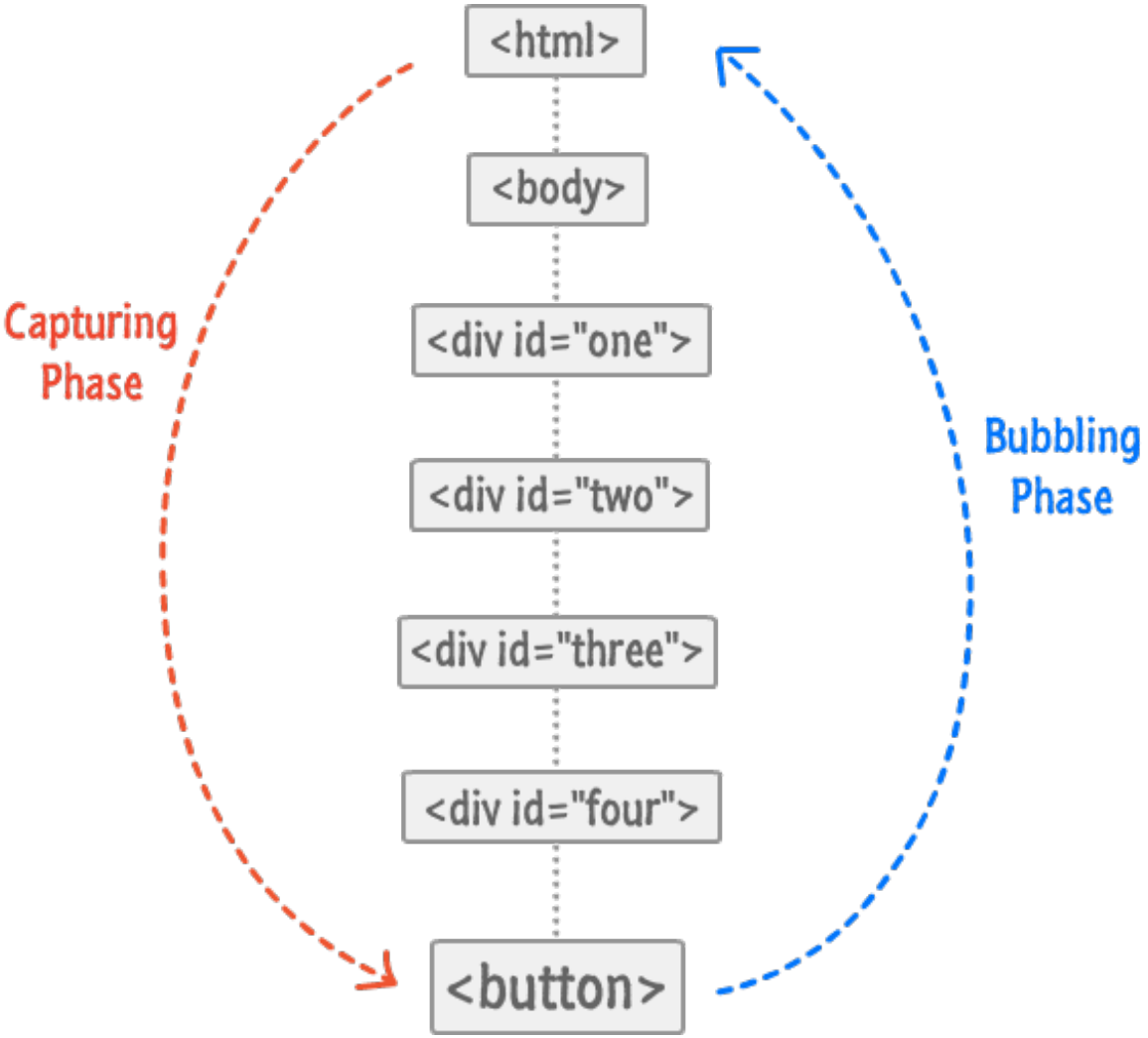
Outline

- Browsers and web application model
- The Window object: setTimeOut, alert etc.
- Event Handling in Modern Browsers
- **Event Propagation in the DOM**

Event Propagation

- Events triggered on an element propagate through the DOM tree in 2 consecutive phases
  - Capture phase: Event is triggered on the topmost element of the DOM and propagates down to the event target element
  - Bubble phase: Event starts from the event target element and 'bubbles up' the DOM tree to the top
- Events may therefore trigger handlers on elements different from their targets

Capture and Bubble Phases



Event Propagation Setup

- To associate an event handler with the capture phase of event propagation, set the third parameter of addEventListener to *true*

```
var div1 = getElementById("one");
div1.addEventListener("click", handler, true);
```

The default way of triggering event handlers is during the bubble phase (3rd argument is false)

Capture Phase

```
var div1 = getElementByID(“one”);
div1.addEventListener(“click”, handler1, true);
var div2 = getElementByID(“two”);
div2.addEventListener(“click”, handler2, true);
```

Assume that the div element ‘two’ is clicked.

handler1 is invoked before handler2 as both are registered during the capture phase.

Bubble Phase

```
var div1 = getElementById("one");
div1.addEventListener("click", handler1, false);
var div2 = getElementById("two");
div2.addEventListener("click", handler2, false);
```

Assume that the div element 'two' is clicked. handler2 is invoked before handler1 as they are both registered during the bubble phase.

Stopping Event Propagation

- In the prior example, suppose handler1 and handler2 are registered in the capture phase

```
var handler1 = function( clickEvent ) {
    clickEvent.stopPropagation();
}
```

Then handler2 will never be invoked as the event will not be sent to div2 in the capture phase

stopPropagation, preventDefault and stopImmediatePropagation

- An event handler can stop the propagation of an event through the capture/bubble phase using the event.*stopPropagation* function
  - Other handlers registered on the element are still invoked however
- To prevent other handlers on the element from being invoked and its propagation, use event.*stopImmediatePropagation*
- To prevent the browser's default action, call the method *event.preventDefault*

Class Activity

- Consider the sample code given in the Github. In what order are the messages in the event handler functions displayed ?

- If you wanted to stop the event propagation in the bubble phase beyond div3, how will you do it ?

Outline

- Browsers and web application model
- The Window object: setTimeOut, alert etc.
- Event Handling in Modern Browsers
- Event Propagation in the DOM