Lecture 5: Closures, Window Object and Event Handling

Building Modern Web Applications
Vancouver Summer Program 2018 (Package E)

Karthik Pattabiraman, Julien Gascon-Samson

The Univerity of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada

Electrical and Computer Engineering

Thursday, July 26, 2018 | Friday, July 27, 2018

Closures

Nested Functions: Closures

- In JavaScript, functions can nest inside other functions, unlike in languages like Java
- Nested functions can access their enclosing function's variables and named arguments (this is a good thing)
- However, nested functions cannot access the parent function's this and variadic arguments (i.e., arguments array)

Closures

- A closure is a nested function that "remembers" the value of it's enclosing function's variables
- Can be used for implementing simple, stateful objects
  - Allow variables to be hidden from other objects
  - Can allow objects to be constructed in parts

Closures: Example





```
1  function Adder(val) {
2     var value = val;
3
4     return function(inc) {
5                         // Returns a function that needs
6                         // to be invoked to get it to
7                         // perform the operation
8
9        value = value + inc;
10                        // Can access parent function
11                        // (Adder)'s local variable
12
13       return value;
14    }
15 };
16
17 var f = Adder(5);
18 document.writeln( f(3) ); // Prints 8
19 document.writeln( f(2) ); // Prints 10
```

Another Example of Closures


```
1  function Counter( initial ) {
2      var val = initial;
3      return {
4          increment: function() { val += 1; },
5          reset: function() { val = initial; }
6          get: function() { return val; }
7      }
8  };
9
10 var f = Counter(5), g = Counter(10);
11 f.increment();  f.reset(); f.increment();
12 g.increment(); g.increment();
13 console.log( f.get() + " , " + g.get() );
```

Why closures are useful ?

- Allow you to remember state in Web Applications
  - Especially when you have many different handlers construct parts of an object (e.g., AJAX messages)
  - Very useful for callbacks in JavaScript: return the callback function from the parent function
  - Way to emulate private variables (JS has none)
- Closures are extensively used in frameworks such as **jQuery** to protect the integrity of internal state

Closures: Referencing Parent Object

- In a closure, what does *this* refer to ?
  - The nested function scope
- But what if you wanted to access the parent function's context (e.g., to invoke a method) ?
  - You no longer get access to parent's *this*
  - Store the parent context in a local variable *that*
- Caution: Can lead to high memory consumption

Referencing Parent Object: Example


```
1  var MultiCounter = {
2     create: function(initial) {
3
4        var that = this;
5        var val = [];
6        console.log(this);
7        this.init = function() {
8           val = [];
9           for (var i=0; i<initial.length; i++) {
10             val.push(  initial[i] );
11          };
12       };
13       this.init();
14       return {
15          increment: function(i) { val[i] += 1; },
16          resetAll: function() { that.init(); },
17          getValues: function() { return val; }
18       };
19    }
20 };
21 var m = MultiCounter.create( [1, 2, 3] );
```

Class Activity- 1


```
1  /* 1) Assume that you want to maintain an array of Counter
       closures, each starting from a different number 1, 2, 3
       etc. Why would the following code not work. Explain why
       not. */
2
3  var MakeCounters1 = function(n) {
4     var counters = [];
5     for (var i=0; i<n; i++) {
6        var val = i;
7        counters[i] = {
8           increment: function() { val++; },
9           get: function() { return val; },
10          reset: function() { val = i; }
11       }
12    }
13    return counters;
14 }
15 var m = MakeCounters1(10);
16 for (var i=0; i<10; i++) {
17    document.writeln("Counter[ " + i + "] = " + m[i].get());
18 }
```

Class Activity- 1- solution


```
1  /* 1- SOLUTION: There is only one instance of the val
       variable. Therefore, all three counter functions will
       set and alter the unique instance of val. Initially, all
        counters will then be set to value "n-1", as this is
       the initial value of the last counter. */
2
3  var MakeCounters1 = function(n) {
4      var counters = [];
5      for (var i=0; i<n; i++) {
6          var val = i;
7          counters[i] = {
8              increment: function() { val++; },
9              get: function() { return val; },
10             reset: function() { val = i; }
11         }
12     }
13     return counters;
14 }
15 var m = MakeCounters1(10);
16 for (var i=0; i<10; i++) {
17     document.writeln("Counter[ " + i + "] = " + m[i].get());
18 }
```

Class Activity- 2


```
1  /* 2) How would you change the code to maintain an array of
       counters the right way (with distinct values from 1 to n
       )? (same code below) */
2
3  var MakeCounters1 = function(n) {
4     var counters = [];
5     for (var i=0; i<n; i++) {
6        var val = i;
7        counters[i] = {
8           increment: function() { val++; },
9           get: function() { return val; },
10          reset: function() { val = i; }
11       }
12    }
13    return counters;
14 }
15 var m = MakeCounters1(10);
16 for (var i=0; i<10; i++) {
17    document.writeln("Counter[ " + i + "] = " + m[i].get());
18 }
```

Class Activity- 2- incorrect solution


```
1  /* 2- BAD SOLUTION: There is only one instance of the loop
       variable i. After the loop is done executing, when
       counters is returned, i==n (this is the stop condition
       of the loop). Therefore, all 3 functions of the counter
       object will try to access element i (n) of val, which
       doesn't exist -- therefore, undefined is returned.*/
2
3  var MakeCounters1a = function(n) {
4      var counters = [];
5      var val = [];
6      for (var i=0; i<n; i++) {
7          val.push(i);
8          counters[i] = {
9              increment: function() { val[i]++; },
10             get: function() { return val[i]; },
11             reset: function() { val[i] = i; }
12         }
13     }
14     return counters; }
15 var m = MakeCounters1a(10);
16 for (var i=0; i<10; i++) {
17     document.writeln("Counter[ " + i + "] = " + m[i].get());
           }
```

Class Activity- 2 (Again)


```
1  /* 2) Going back to the question -- how would you change the
        code to maintain an array of counters the right way (
       with distinct values from 1 to n), without creating a
       val array? */
2
3  var MakeCounters1 = function(n) {
4     var counters = [];
5     for (var i=0; i<n; i++) {
6        var val = i;
7        counters[i] = {
8           increment: function() { val++; },
9           get: function() { return val; },
10          reset: function() { val = i; }
11       }
12    }
13    return counters;
14 }
15 var m = MakeCounters1(10);
16 for (var i=0; i<10; i++) {
17    document.writeln("Counter[ " + i + "] = " + m[i].get());
18 }
```

Class Activity- 2 - solution and 3 - optimization


```
1  /* 3) In class activity 2, did you end up adding additional
       fields to counters? If so, then can you come up with a
       different solution without such additional fields?
2     Why do we need the "this" keyword?*/
3
4  var MakeCounters2 = function(n) {
5     var counters = [];
6     for (var i=0; i<n; i++) {
7        counters[i] = {
8           val : i,
9           initial : i,
10          increment: function() { this.val++; },
11          get: function() { return this.val; },
12          reset: function() { this.val = this.initial; }
13       }
14    }
15    return counters;
16 }
17 var m = MakeCounters2(10);
18 for (var i=0; i<10; i++) {
19    document.writeln("Counter[ " + i + "] = " + m[i].get());
20 }
```

Class Activity- 3 - solution


```
1  /* 4) What's the advantage of this solution compared to the
       previous one?
2     Why don't we need the "this" keyword?*/
3
4  var MakeCounters3 = function(n) {
5     var counters = [];
6     for (var i=0; i<n; i++) {
7        counters[i] = function( ) {
8           var initial = i, val = initial;
9           return {
10             increment: function() { val++; },
11             get: function() { return val; },
12             reset: function() { val = initial; }
13          }
14       }();  // Why do we need the parentheses () ?
15    }
16    return counters;
17 }
18 var m = MakeCounters3(10);
19 for (var i=0; i<10; i++) {
20    document.writeln("Counter[ " + i + "] = " + m[i].get());
21 }
```

Gotchas with Closures

- Remember, the closure stores a link to the variables of the original function, not a copy
  - Any changes to the enclosing variable are reflected in the closure, even after it was created
- Keep the amount of state you want to save in the closure to the minimum necessary state
  - Otherwise, garbage collector cannot release it and you will get memory leaks, and run out of memory

Window Object

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

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 (2)

- Browser follows two phase execution model

Phase 2

- Waits for events to be triggered 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

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

Example

```
1  do {
2      var name = prompt("What is your name?");
3      var correct = confirm("You entered: " + name);
4  } while (! correct);
5  // This is bad security practice - don't do this !
6  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 clearInterval is invoked

Example of setTimeout

```
1  var timeoutHandler = function(message) {
2      return function() {
3          alert(message);
4      };
5  };
6
7  var ret = setTimeout(timeoutHandler("Hello"),100);
8  // [...]
9  if (flag) clearTimeout(ret);
```

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 clearInterval is invoked

Example of setInterval

```
1  var intervalHandler = function(message) {
2      var i = 0;
3      return function() {
4          alert(message + '  ' + i);
5          i+ = 1;
6      }
7  };
8  var ret = setInterval(intervalHandler("invocation"
       ),1000); // [...]
9  if (flag) clearInterval(ret);
```

Class Activity

- Create a new function that invokes another function func a specified number of times noTimes, asynchronously, each time after time ms.
- The function should pass as an argument to func the number of times it called func so far.

HINT

You can do it through setTimeout or setInterval

```
1  function invokeTime( func, noTimes, time) {
2      // ...
3  }
4  var setup = function() {
5      invokeTimes( function(i) { alert("hello " + i); }, 10,
           1000 );
6  }
7
8  setup();
```

Class Activity – Solution using setInterval





```
1  var invokeTimes = function(func, noTimes, time) {
2     console.log("Setting up interval " + noTimes + " " + time);
3     var count = 0;
4     var interval;
5     var intervalHandler = function() {
6        console.log( "invocation " + count);
7        func(count);
8        count = count + 1;
9        if (count == noTimes) {
10          clearInterval(interval);
11       }
12    };
13    if (noTimes >0)
14       interval = setInterval(intervalHandler, time);
15 };
16
17 var setup = function() {
18    invokeTimes( function(i) { alert("hello " + i); }, 10, 1000 );
19 }
20
21 setup();
```

Class Activity – Solution using setTimeout





```
1  var invokeTimes = function(func, noTimes, time) {
2      console.log("Setting up interval " + noTimes + " " + time);
3      var count = 0;
4      var timeoutHandler = function() {
5          // timeOutHandler is a closure
6          console.log( "invocation " + count);
7          func(count);
8          count = count + 1;
9          if (count < noTimes) {
10             setTimeout(timeoutHandler, time);
11         }
12     };
13     if (count==0) setTimeout(timeoutHandler, time);
14 };
15
16 var setup = function() {
17     invokeTimes( function(i) { alert("hello " + i); }, 10, 1000 );
18 }
19
20 setup();
```

Other Solution using setTimeout and closures


```
1  // We need to use nested closures to preserve the state here
2
3  var invokeTimes = function(func, noTimes, time) {
4     console.log("Setting up interval " + noTimes + " " + time);
5
6     for (var i = 0; i < noTimes; ++i) {
7        var timeoutHandler = function(count) {
8           // timeOutHandler is a closure
9           return function() {
10             console.log( "invocation " + count);
11             func(count);
12          }
13       }
14       setTimeout(timeoutHandler(i), time * i);
15    }
16 };
17
18 var setup = function() {
19    invokeTimes( function(i) { alert("hello " + i); }, 5, 1000 );
20 }
21
22 setup();
```

Event Handling in Modern Browsers

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

```
1  element.onclick = function(event) {
2      this.style.backgroundcolor = "#ffffff";
3      return true;
4  }
```

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

```
1  var old = element.onclick;
2  element.onclick = function(event) {
3      this.style.backgroundcolor = "#ffffff";
4      if (old) return old(event);
5      return true;
6  }
```

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

Example

```
1  var b = document.getElementById("mybutton");
2  b.addEventListener("click", function() {
3          alert("hello");
4      }, 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)

Example

```
1  var b = document.getElementById("mybutton");
2  b.addEventListener("click", function() {
3          alert("hello");
4      }, false);
5  b.addEventListener("click", function() {
6          alert("world");
7      }, 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

Example

```
1  var handleMouseclick = function() {
2          alert("clicked");
3      };
4  var b = document.getElementById("mybutton")
5  b.addEventListener("click", handleMouseClick,
       false);
6  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




- Consider an HTML page containing three buttons having id reset, increment and done (use above "H" icon to see the HTML code)
- Write **a single handler function** (i.e., popup) for the click property of each of the three buttons that displays a message (str1 + str2) using the JavaScript alert function
  - 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.

```
1  window.onload = function() {
2      var setupBtn = document.getElementById("reset");
3      var runBtn = document.getElementById("increment");
4      var doneBtn = document.getElementById("done");
5      setupBtn.addEventListener( "click", /* ??? */, false);
6      runBtn.addEventListener( "click", /* ??? */, false);
7      doneBtn.addEventListener( "click", /* ??? */, false); }
```

Event Propagation in the DOM

Event Propagation

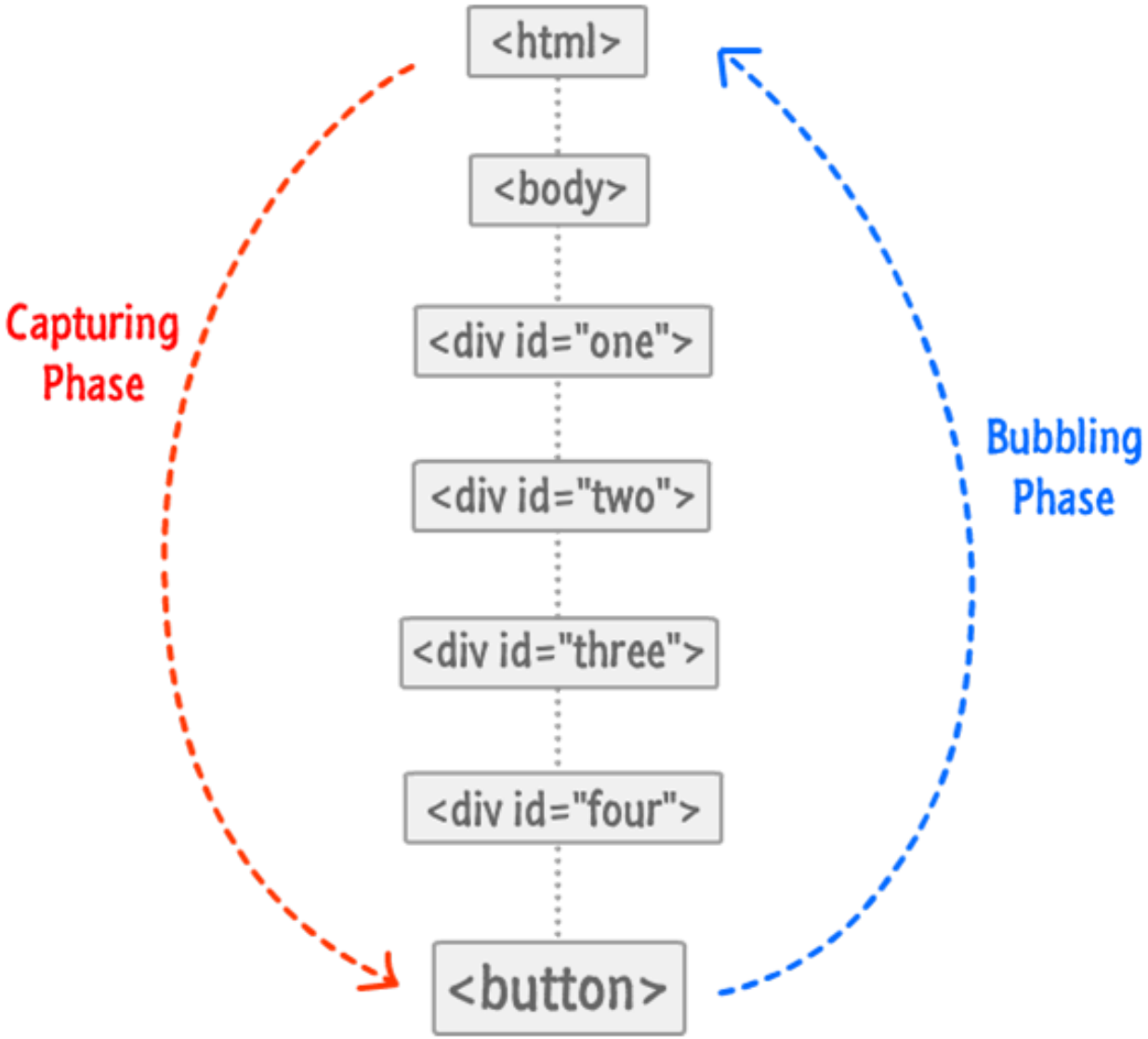
- Events triggered on an element propagate through the DOM tree in 2 consecutive phases
  - Capture phase: Event is triggered on the topmost element of the DOM and propagates down to the event target element
  - Bubble phase: Event starts from the event target element and 'bubbles up' the DOM tree to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

Exception: for the target element itself

- For the target element itself, the W3C standards considers a *target* phase
- All handlers registered for the target element are always registered for the target phase – the bubble/capture phase argument is ignored when registering handlers (see later)

- 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

Example

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

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

Capture and Bubble Phases


```
1  var div1 = getElementByID("one");
2  div1.addEventListener("click", handler1, true);
3  var div2 = getElementByID("two");
4  div2.addEventListener("click", handler2, true);
```

Capture Phase

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

Bubble Phase

- 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

Stopping Event Propagation

```
1  var handler1 = function( clickEvent ) {
2      clickEvent.stopPropagation();
3  }
```

- 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 JS sample code in prop.js (click on the JS link at the top-right to open it). 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 ?

Table of Contents