DOM and Events - Part 2

Building Modern Web Applications - VSP2019

Karthik Pattabiraman
Kumseok Jung

Modern Browsers

1. **Modern Browsers**
2. DOM Event Handling
3. DOM Event Propagation

Modern Browsers: 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 Browsers: 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

Modern Browsers: JavaScript Execution Model

- 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

Modern Browsers: 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.

Modern Browsers: window Object

- `alert`: Simple way to pop-up 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

```
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);
```

Modern Browsers: window Object

- `setTimeout` is used to schedule a future event asynchronously once after a specified number of milliseconds (can be set to 0)
  - Can specify arguments to event handler
  - Can be cancelled using the `clearTimeout` method


```
1  var callback = function(){
2     alert("Hello");
3  }
4  var timer = setTimeout(callback, 1000);
5
6  clearTimeout(timer);
```

Modern Browsers: window Object

- `setTimeout` is used to schedule a future event asynchronously once after a specified number 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

```
1  var count = 0;
2  var callback = function(){
3     alert("Hello " + (count++));
4  }
5  var timer = setInterval(callback, 1000);
6  clearInterval(timer);
```

Class Activity

lectures/lecture-5/interval.html
lectures/lecture-5/interval.js

- Create a new function that invokes another function `func` a specified number of times `num`, 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 invokeTimes(func, num, time){
2     // to implement
3  }
4
5  invokeTimes(function(count){
6     alert("Hello " + count);
7  }, 10, 1000);
```

Class Activity: Solution 1

lectures/lecture-5/interval.html
lectures/lecture-5/interval.js

```
1  // Using setInterval
2  function invokeTimes(func, num, time){
3     var count = 0;
4     var interval;
5     var intervalHandler = function(){
6        func(count);
7        count += 1;
8        if (count === num) clearInterval(interval);
9     };
10    if (num > 0) interval = setInterval(intervalHandler, time);
11 }
12
13 invokeTimes(function(count){
14    alert("Hello " + count);
15 }, 10, 1000);
16
17
```

Class Activity: Solution 2

lectures/lecture-5/interval.html
lectures/lecture-5/interval.js

```
1  // Using setTimeout
2  function invokeTimes(func, num, time){
3     var count = 0;
4     var timeoutHandler = function(){
5        func(count);
6        count += 1;
7        if (count < num) setTimeout(timeoutHandler, time);
8     };
9     if (num > 0) setTimeout(timeoutHandler, time);
10 }
11
12 invokeTimes(function(count){
13    alert("Hello " + count);
14 }, 10, 1000);
15
16
17
```

DOM Event Handling

1. Modern Browsers
2. **DOM Event Handling**
3. DOM Event Propagation

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)

Event Handling: Cautionary Note

- 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)

Event Handling: 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. e.g., `onload`, `onmousemove`

```
1  var elem = document.getElementById("mybutton");
2  element.onclick = function(event){
3     this.style.backgroundColor = "#fff";
4     return true;
5  };
```

1. `this` is bound to the DOM element on which the `onclick` handler is defined – can access its properties through `this[prop]`
2. `return` value of `false` tells browser not to perform the default value associated with the property (`true` otherwise)

Registering Event Handlers: DOM 2.0

- The DOM 1.0 method is clunky and can be buggy. Also, difficult to remove event handlers
- DOM 2.0 event handlers
  - `addEventListener` for adding a event handler
  - `removeEventListener` for removing event handlers
  - `stopPropagation` and `stopImmediatePropagation` for stopping the propagation of an event

DOM 2.0: 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 by default

```
var elem = document.getElementById(“mybutton”);
elem.addEventListener(“click”, function(event){
   this.style.backgroundColor = “#fff”;
   return true;
});
```

DOM 2.0: 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)

```
1  var elem = document.getElementById(“mybutton”);
2  elem.addEventListener(“click”, function(event){
3     alert(“Hello”);
4  });
5  elem.addEventListener(“click”, function(event){
6     alert(“World”);
7  });
```

DOM 2.0: 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

```
1  var clickHandler = function(event){
2     alert("Clicked");
3  };
4  var elem = document.getElementById("mybutton");
5  elem.addEventListener("click", clickHandler);
6  elem.removeEventListener("click", clickHandler);
7
```

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: Click Events

lectures/lecture-5/domclick.html
lectures/lecture-5/domclick.js

- Consider an HTML containing 3 buttons with ids `reset`, `up`, and `down`
- Write **3 handler functions** for the `click` event of each of the 3 buttons to do the following:
  - `resetBtn` should set the `count` to 0
  - `upBtn` should increment the `count` by 1
  - `downBtn` should decrement the `count` by 1

```
1  window.onload = function(){
2     var count = 0;
3     var resetBtn = document.getElementById("reset");
4     var upBtn = document.getElementById("up");
5     var downBtn = document.getElementById("down");
6     resetBtn.addEventListener("click", /* ??? */);
7     upBtn.addEventListener("click", /* ??? */);
8     downBtn.addEventListener("click", /* ??? */);    };
```

Class Activity: Closures

lectures/lecture-5/multiclick.html
lectures/lecture-5/multiclick.js

Fix the following code - all buttons are showing the same message!

```
1  function addClickListeners (buttons){
2     for (var i = 0; i < buttons.length; i++){
3        buttons.addEventListener("click", function(){
4           alert("Clicked Button " + i);
5        });
6     }
7     return buttons;
8  };
9
10 var btns = document.getElementsByTagName("button");
11 addClickListeners(btns);
12
13
14
```

Class Activity: Closures

Solution: Capture the value of `i` into the scope of the closure function

```
1  function addClickListeners (buttons){
2     for (var i = 0; i < buttons.length; i++){
3        var ownHandler = (function(j){
4           return function(){
5              alert("Clicked Button " + j);
6           }
7        })(i);
8        buttons.addEventListener("click", ownHandler);
9     }
10    return buttons;
11 };
12
13 var btns = document.getElementsByTagName("button");
14 addClickListeners(btns);
```

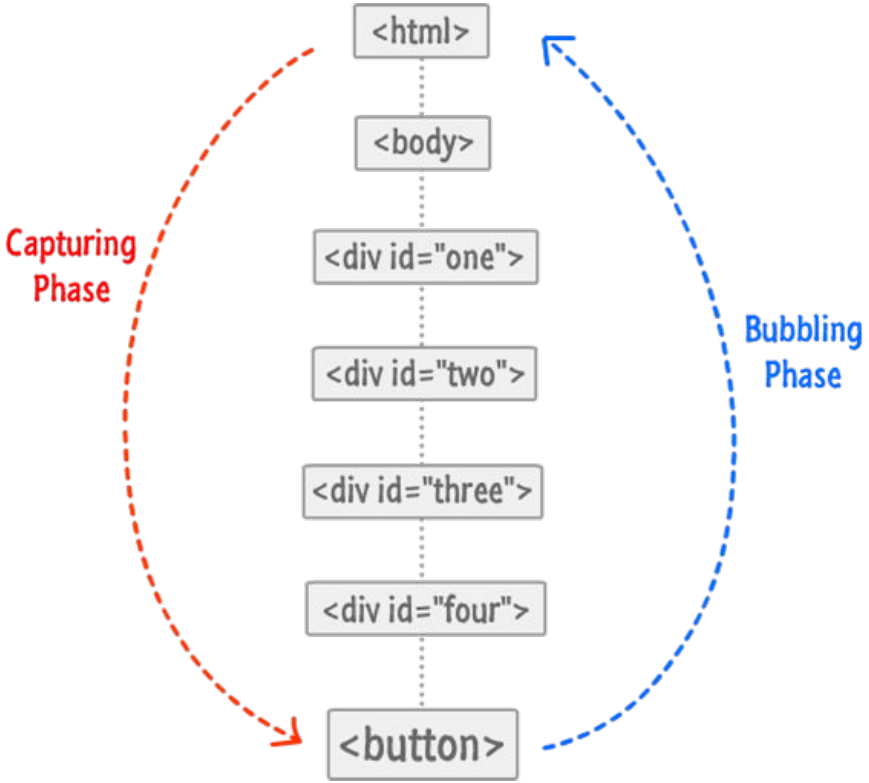
DOM Event Handling

1. Modern Browsers
2. DOM Event Handling
3. **DOM Event Propagation**

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`

```
1  var div1 = document.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 = document.getElementById(“one”);
2  div1.addEventListener(“click”, handler1, true);
3  var div2 = document.getElementById(“two”);
4  div2.addEventListener(“click”, handler2, true);
```

Capture Phase

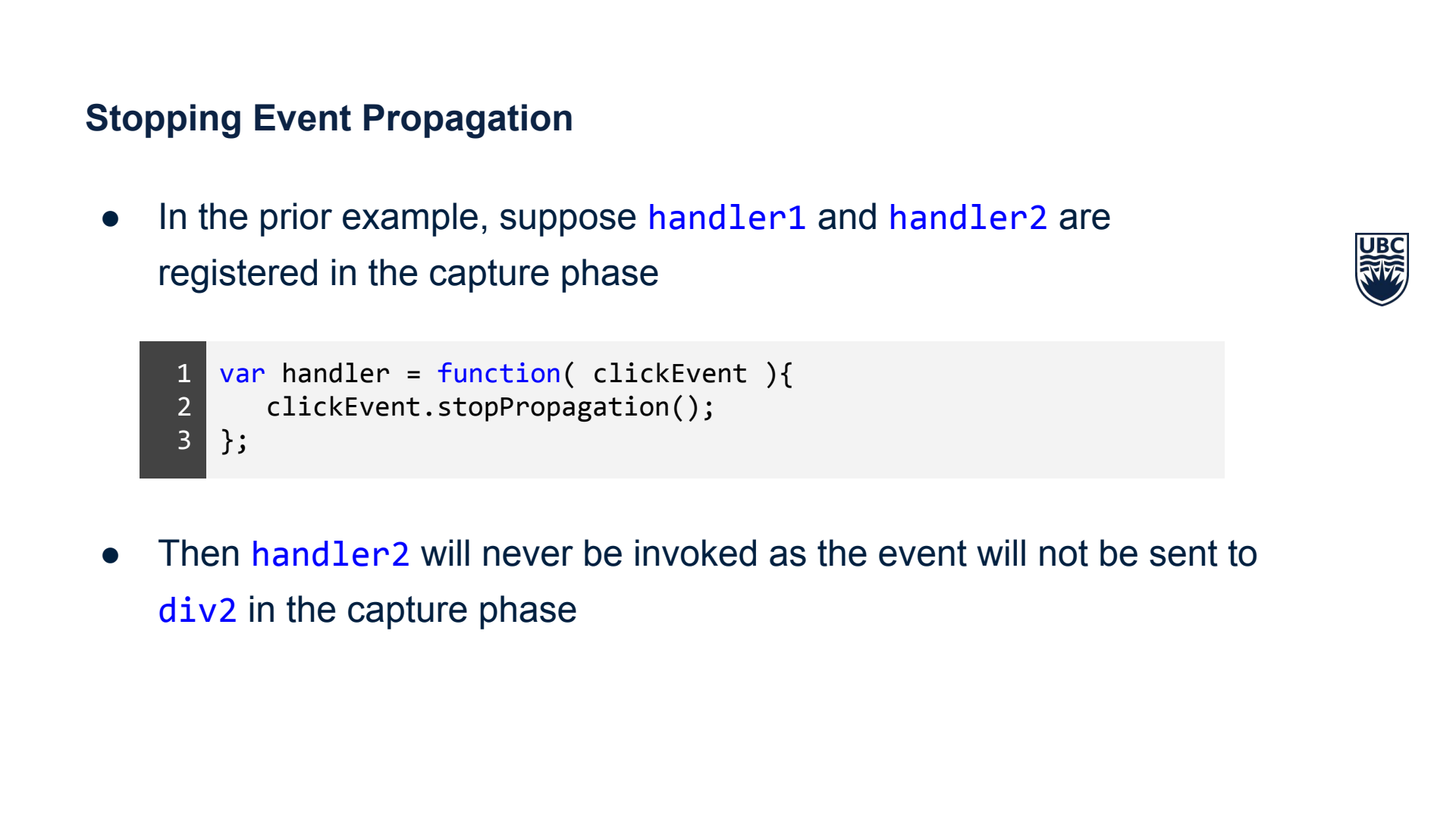
- Assume that the `<div>` element `‘two’` is clicked.
- `handler1` is invoked before `handler2` as both are registered during the capture phase.

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



```
var handler = 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`, `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

lectures/lecture-5/prop.html
lectures/lecture-5/prop.js

- Consider the JS sample code in **prop.js**. 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?