DOM and Events

Building Modern Web Applications - VSP2019

Karthik Pattabiraman
Kumseok Jung

Recap from Lecture 1

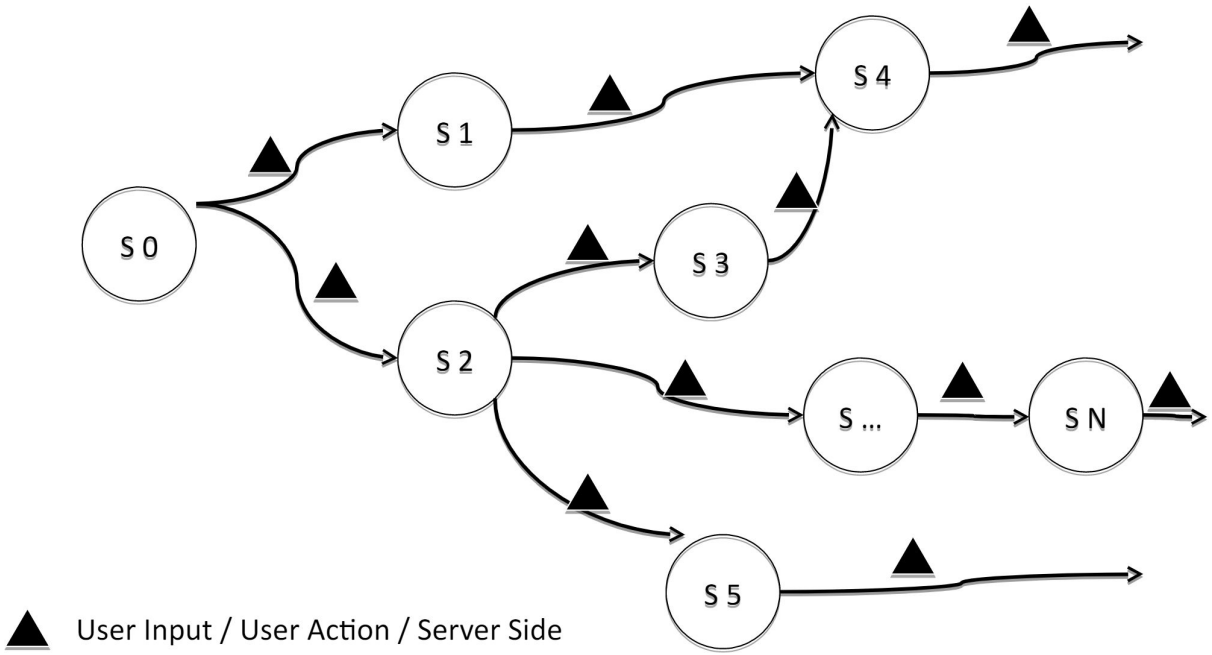
1. **Recap from Lecture 1**
2. DOM APIs
3. DOM Traversal
4. DOM Manipulation
5. DOM Events

Recap from Lecture 1: DOM

- Hierarchical representation of the contents of a web page – initialized with static HTML
- Can be manipulated from within the JavaScript code (both reading and writing)
- Allows information sharing among multiple components of web application

DOM: an evolving entity

DOM is highly dynamic!



Why study DOM interactions?

- Needed for JS code to have any effect on webpage (without reloading the page)
- Uniform API/interface to access DOM from JS
- Does not depend on specific browser platform

NOTE

- We'll be using the native DOM APIs for many of the tasks in this lecture
- Though many of these can be simplified using frameworks such as jQuery, it is important to know what's "under the hood"
- We assume a standards compliant browser!

DOM APIs

1. Recap from Lecture 1
2. **DOM APIs**
3. DOM Traversal
4. DOM Manipulation
5. DOM Events

Selecting HTML Elements

- You can access the DOM from the object `window.document` and traverse it to any node
- However, this is slow – often you only need to manipulate specific nodes in the DOM
- Further, navigating to nodes this way can be error prone and fragile
  - Will no longer work if DOM structure changes
  - DOM structure changes from one browser to another

Selecting HTML Elements

- With a specified `id`
- With a specified tag name
- With a specified `class`
- With generalized CSS selector

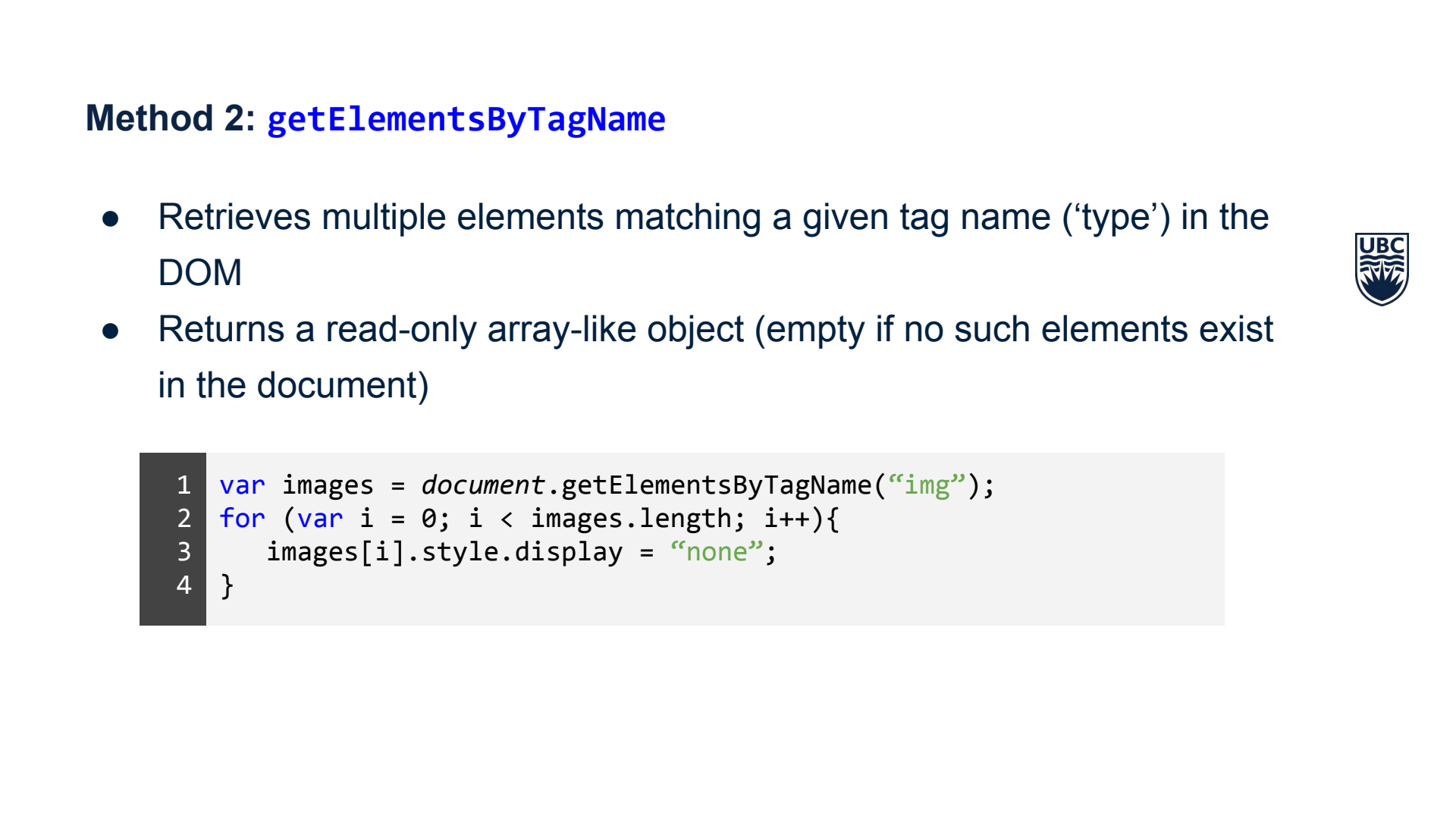
Method 1: `getElementById`

- Used to retrieve a single element from DOM
  - IDs are unique in the DOM (or at least must be)
  - Returns `null` if no such element is found

```
1  var id = document.getElementById(“Section1”);
2  if (id === null) throw new Error(“No element found”);
```

Method 2: `getElementsByTagName`

- Retrieves multiple elements matching a given tag name (‘type’) in the DOM
- Returns a read-only array-like object (empty if no such elements exist in the document)



```
1  var images = document.getElementsByTagName(“img”);
2  for (var i = 0; i < images.length; i++){
3     images[i].style.display = “none”;
4  }
```

Method 3: `getElementsByClassName`

- Can also retrieve elements that belong to a specific CSS class
  - More than one element can belong to a CSS class


```
1 var warnings = document.getElementsByClassName(“warning”);
2 if (warnings.length > 0){
3    console.log(“Found ” + warnings.length + “ elements”);
4 }
```

Important point: Live Lists

- Both `getElementsByClassName` and `getElementsByTagName` return **live lists**
  - List can change after it is returned by the function if new elements are added to the document
  - List cannot be changed by JavaScript code adding to it or removing from it directly though
- Make a copy if you're iterating through the lists

Selecting Elements by CSS selector

- Can also select elements using generalized CSS selectors using `querySelectorAll()` method
  - Specify a selector query as argument
  - Query results are not “live” (unlike earlier)
  - Can subsume all the other methods
- `querySelector()` returns the first element matching the CSS query string, `null` otherwise

CSS selector examples

```
1  “#nav”          // Any element with id=“nav”
2
3  “div”           // Any <div> element
4
5  “.warning”      // Any element with “warning” class
6
7  “#log span”     // Any <span> descendant of id=“log”
8
9  “#log > span” // Any <span> child element of id=“log”
10
11 “body > h1:first-child”     // first <h1> child of <body>
12
13 “div, #log”     // All <div> elements and element with id=“log”
14
```

Invocation on DOM subtrees

- All of the above methods can also be invoked on DOM elements not just the document
  - Search is confined to subtree rooted at element
- Example: Assume element with id=”log” exists

```
1 var log = document.getElementById(“log”);
2 var error = log.getElementsByClassName(“error”);
3 if (error.length === 0){ … }
4
```

Class Activity

lectures/lecture-3/changeImage.html
lectures/lecture-3/changeImage.js

- Assume the page contains a `<div>` element with ID `id`, which contains a series of images (`<img>` nodes)
- Write a function that takes two arguments, `id` and `interval`. At each `interval`, the images must be “rotated”, i.e., `image0` will become `image1`, `image1` will become `image2`, etc.

```
1  function changeImages(id, interval){
2
3  }
```

- To repeat the execution of a given function `f` at a specific interval (e.g. 1000 ms): `setInterval(1000, f);`

DOM Traversal

1. Recap from Lecture 1
2. DOM APIs
3. **DOM Traversal**
4. DOM Manipulation
5. DOM Events

Traversing the DOM

- Since the DOM is just a tree, you can walk it the way you'd do with any other tree
  - Typically using recursion
- Every browser has minor variations in implementing the DOM, so should not be sensitive to such changes
  - Traversing DOM this way can be fragile

Before accessing or manipulating the DOM...

Problem

- When your JS code executes, the page might not have finished loading
  - The DOM tree might not be fully instantiated / might change!

`window.onload`

- Event that gets fired when the DOM is fully loaded (we'll get back to events later...)
- You can give a callback function to execute upon proper loading of the DOM.
- Your DOM manipulation code should go inside that function

```
1 // Using DOM Level 1 API -- not recommended
2 window.onload = function(){ /* Access the DOM here */ }
```

Properties for DOM Traversal

- `parentNode`: Parent node of this one, or `null`
- `childNodes`: A read only array-like object containing all the (live) child nodes of this one
- `firstChild`, `lastChild`: The first and last child of a node, or `null` if it has no children
- `nextSibling`, `previousSibling`: The next and previous siblings of a node (in the order in which they appear in the document)

Other node properties

- `nodeType`: ‘kind of node’
  - Element node: 1
  - Text node: 3
  - Comment node: 8
  - Document node: 9
- `nodeValue`: Textual content of Text of comment node
- `nodeName`: Tag name of a node, converted to upper-case

Exercise: Find a Text Node

- We want to find the DOM node that has a certain piece of text, say “text”
- Return `true` if text is found, false otherwise
- We need to recursively walk the DOM looking for the text in all text nodes


```
1  function search(node, text){
2     /* ... */
3  };
4  var result = search(window.document, “Hello world!”);
```

Exercise: Find a Text Node

Solution:

```
1  function search(node, text){
2     if (node.nodeType === 3 && node.nodeValue === text){
3        return true;
4     }
5     else if (node.childNodes){
6        for (var i = 0; i < node.childNodes.length; i++){
7           var found = search(node.childNodes[i], text);
8           if (found) return found;
9        }
10    }
11    return false;
12 };
13 var result = search(window.document, “Hello world!”);
```

Class Activity

lectures/lecture-3/domcollect.html
lectures/lecture-3/domcollect.js

- Write a function that will traverse the DOM tree rooted at a node with a specific id, and **checks if** any of its **sibling nodes** and **itself** in the document **is a text node**, and if so, concatenates their text content and returns it.
- Can you generalize it so that it works for the entire subtree rooted at the sibling nodes?

DOM Manipulation

1. Recap from Lecture 1
2. DOM APIs
3. DOM Traversal
4. **DOM Manipulation**
5. DOM Events

Adding and removing nodes

- DOM elements are also JavaScript Objects (in most browsers) and consequently can have their properties read and written to
  - Can extend DOM elements by modifying their prototype objects
  - Can add fields to the elements for keeping track of state (e.g., visited node during traversals)
  - Can modify HTML attributes of the node such as width etc. – changes reflected in browser display

Creating New and Copying Existing DOM Nodes

- Creating New DOM Nodes
  - Using either *document*.createElement(“element”) OR *document*.createTextNode(“text content”)

```
1 var newNode = document.createTextNode(“hello”);
2 var elNode = document.createElement(“h1”);
```

- Copying Existing DOM Nodes: use cloneNode
  - Single argument can be true or false
    - True: deep copy (recursively copy all descendants)
  - new node can be inserted into a different document

```
1 var existingNode = document.getElementById(“my”);
2 var newNode = existingNode.cloneNode(true);
```

Inserting Nodes

- `appendChild`: Adds a new node as a child of the node it is invoked on. node becomes `lastChild`
- `insertBefore`: Similar, except that it inserts the node before the one that is specified as the second argument (lastChild if it's `null`)

```
1 var s = document.getElementById("my");
2 s.appendChild(newNode);
3 s.insertBefore(newNode, s.firstChild);
```

Removing and replacing nodes

- Removing a node `n`: `removeChild`

```
1  n.parentNode.removeChild(n);
```

- Replacing a node `n` with a new node: `replaceChild`

```
1  var edit = document.createTextNode("[redacted]");
2  n.parentNode.replaceChild(edit, n);
```

Class Activity

lectures/lecture-3/domadd.html
lectures/lecture-3/domadd.js

- Write a function `newdiv` that takes two parameters: a node `n` and a string `id`. The function should replace node `n` by making it a child of a new `<div>` element with id = `id`.

```
1  /* function to replace a node n by making it a child of a new
2     <div> element with id = “id” */
3  function newdiv(n, id){
4     /* ... */
5  };
```

Class Activity

lectures/lecture-3/domadd.html
lectures/lecture-3/domadd.js

- Write a function `newdiv` that takes two parameters: a node `n` and a string `id`. The function should replace node `n` by making it a child of a new `<div>` element with id = `id`.


```
1  /* function to replace a node n by making it a child of a new
2     <div> element with id = “id” */
3  function newdiv(n, id){
4     var div = document.createElement(“div”);
5     div.id = id;
6     n.parentNode.replaceChild(div, n);
7     div.appendChild(n);
8  };
```

DOM Events

1. Recap from Lecture 1
2. DOM APIs
3. DOM Traversal
4. DOM Manipulation
5. **DOM Events**

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

lectures/lecture-3/domclick.html
lectures/lecture-3/domclick.js

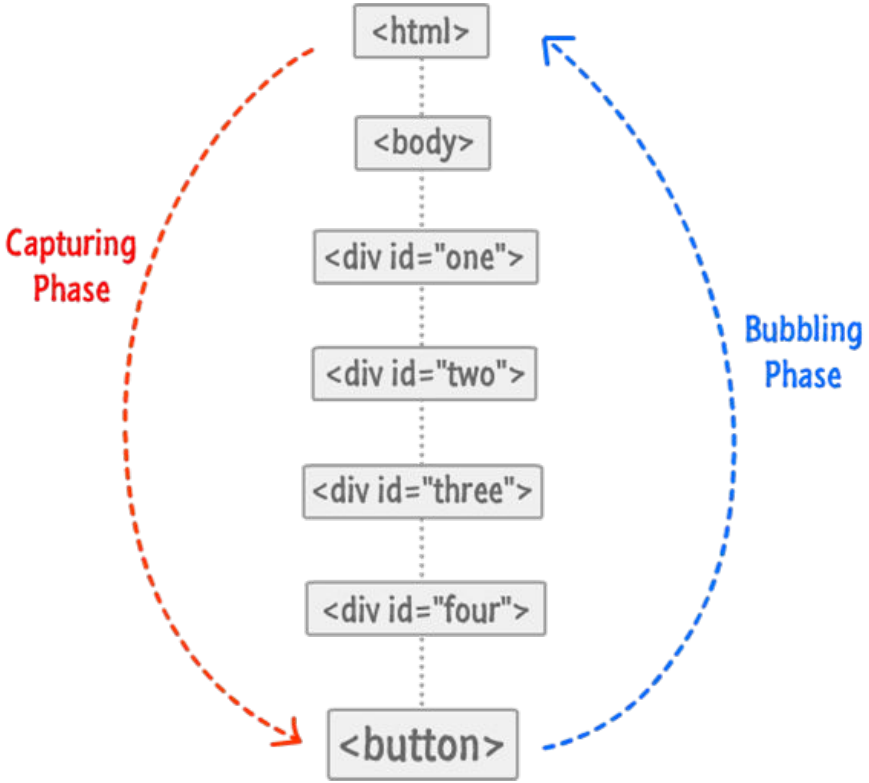
- Consider an HTML containing 3 buttons with ids `reset`, `increment`, and `done`
- Write **a single handler function** (i.e., `popup`) for the `click` event of each of the 3 buttons to display a message (`str1` + `str2`) using the `alert` function
  - `str1` is determined at runtime **when setting the 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", /* ??? */);
6     runBtn.addEventListener("click", /* ??? */);
7     doneBtn.addEventListener("click", /* ??? */);    };
```

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

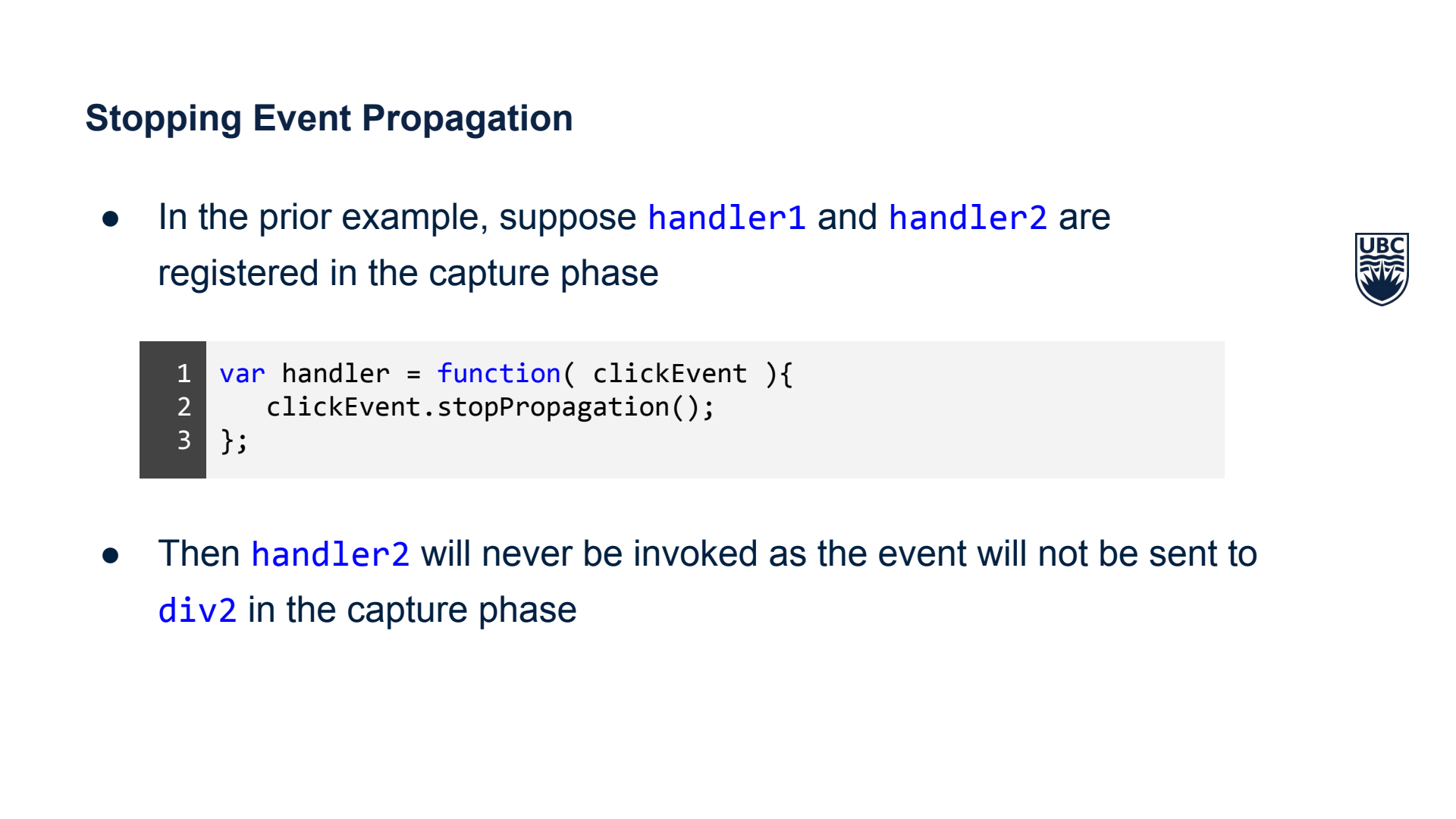
- Assume that the `<div>` element `‘two’` is clicked.
- `handler1` is invoked before `handler2` as both are registered during the capture phase.

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-3/prop.html
lectures/lecture-3/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?