Revision

Building Modern Web Applications - VSP2019

Karthik Pattabiraman
Kumseok Jung

HTML and CSS

1. **HTML and CSS**
2. DOM and Events
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. Exam Logistics

Web Applications: What are they?

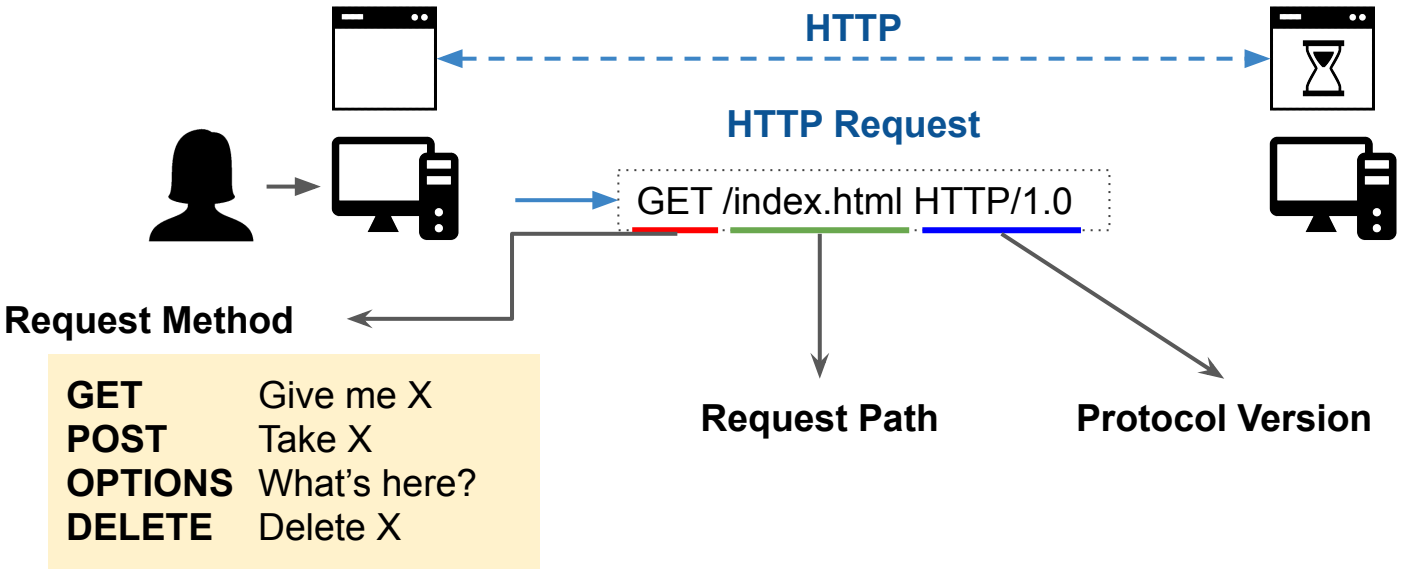
Desktop Application	Web Application
Connection to internet **not required**	Connection to internet **required**
Processing on **local device** only	Processing on **local device (client)** and **remote device (server)**
Software **delivered via storage medium**	Software **delivered via network**
Software **installed to the local OS**	Software **interpreted by the browser**
Can run on **local device** only	Can run **from any device**

HTTP (HyperText Transfer Protocol)

- Application layer protocol for exchanging HyperText documents (and others)
  - A **standard** defining how web client and web server should exchange information
- HTTP Request
  - Defines the message format a **client** should follow
- HTTP Response
  - Defines the message format a **server** should follow

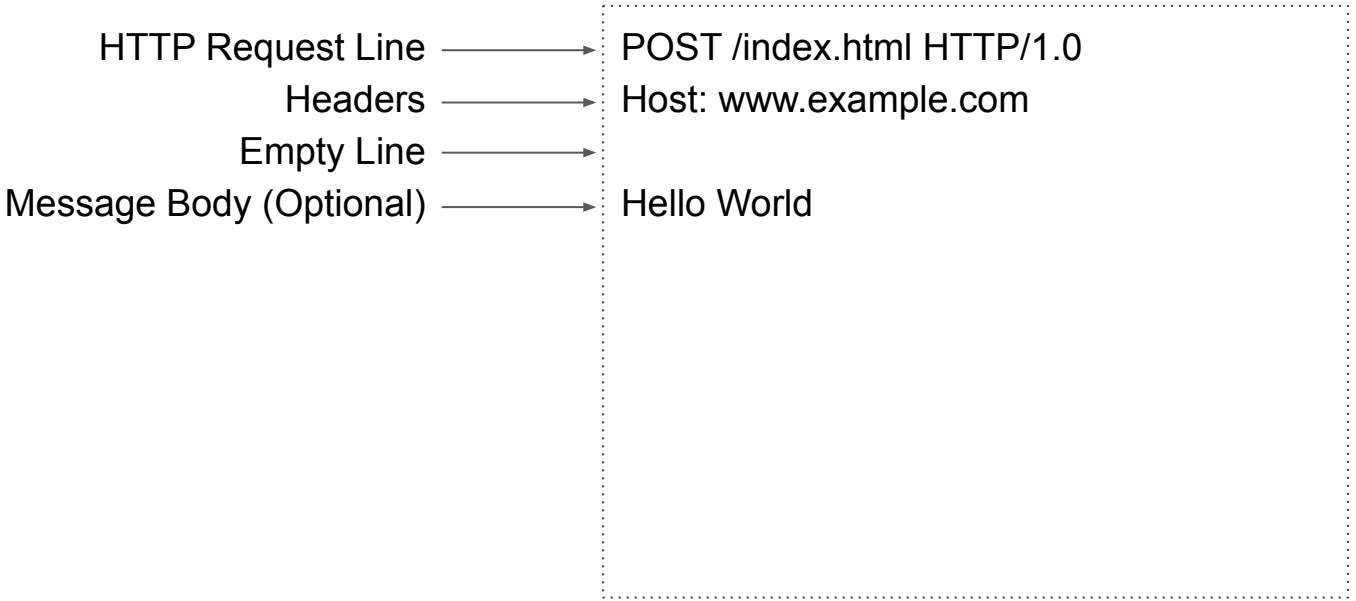
HTTP (HyperText Transfer Protocol)

- HTTP Request
  - Defines the message format a **client** should follow



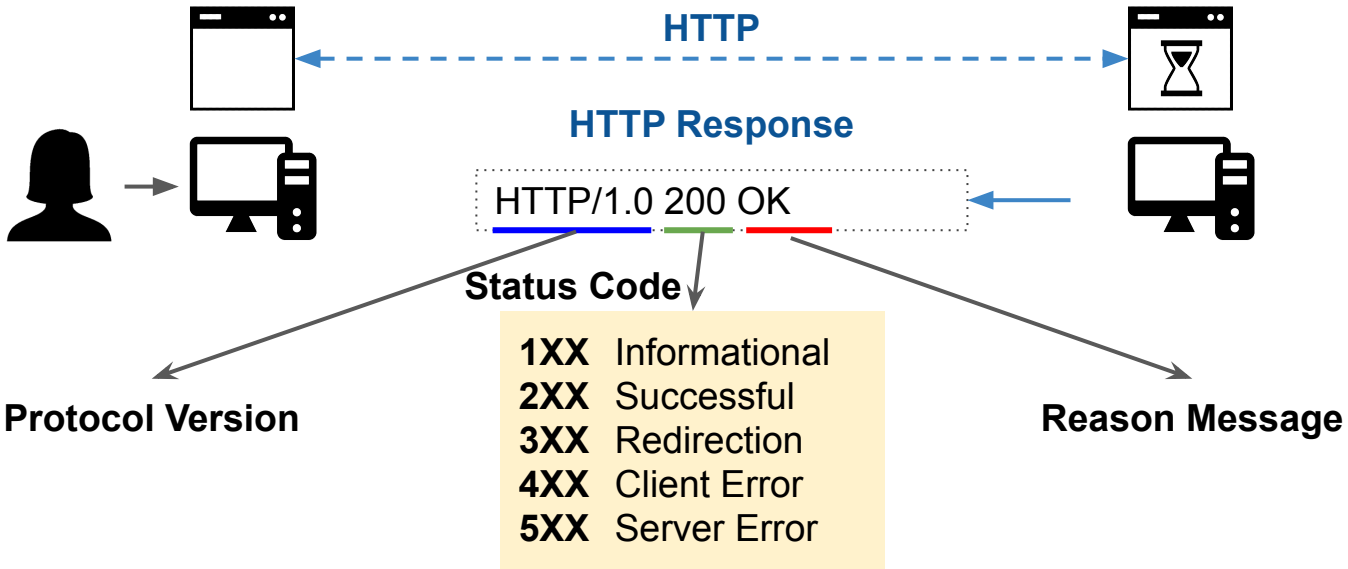
HTTP (HyperText Transfer Protocol)

- HTTP Request
  - Defines the message format a **client** should follow



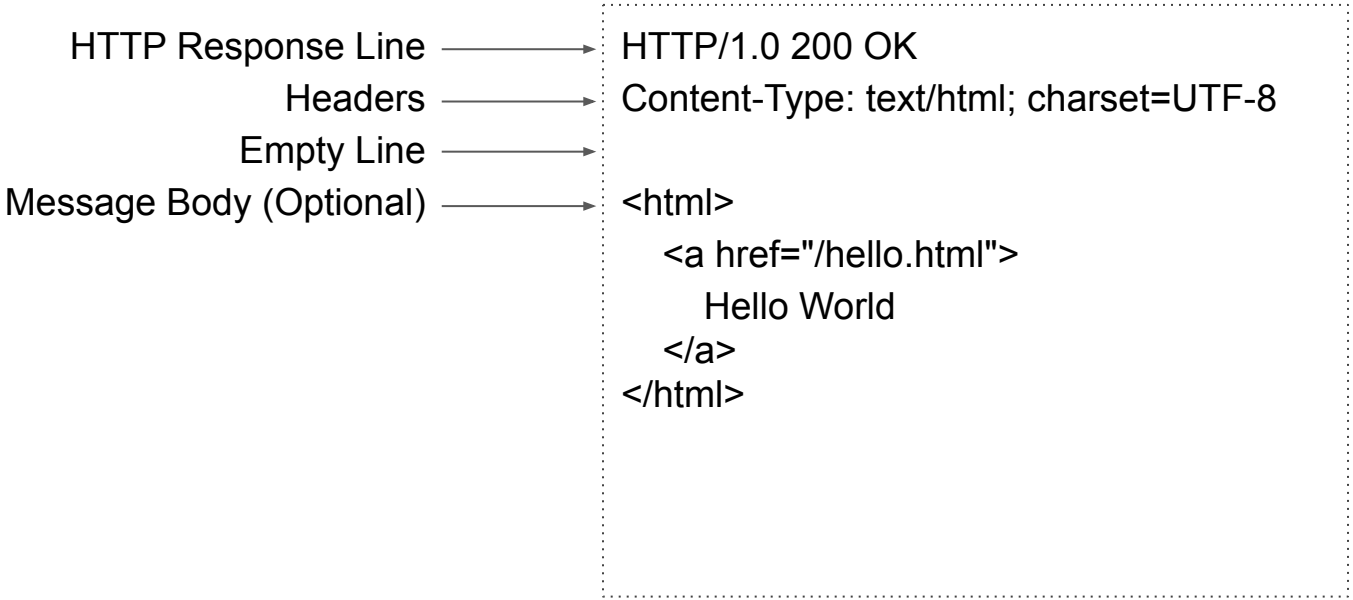
HTTP (HyperText Transfer Protocol)

- HTTP Response
  - Defines the message format a **server** should follow



HTTP (HyperText Transfer Protocol)

- HTTP Response
  - Defines the message format a **server** should follow



HTTP and HTML: The beginnings

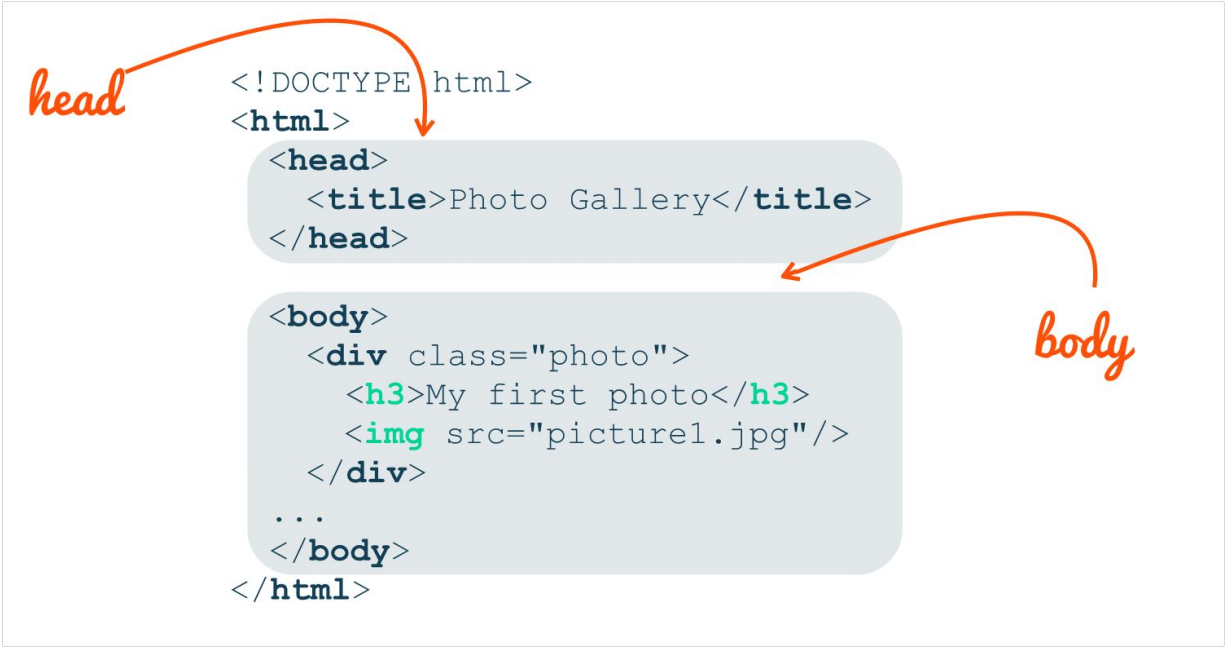
- **3 essential components** of a web application
  - **Server**: To "serve" the web-page and to send content to the client
  - **Client**: To receive content from the server and display them on the web browser window
  - **HTTP connection** for client-server interactions
- Everything else is optional

HTML (HyperText Markup Language)

- Hypertext Markup Language to describe the structure and contents of the initial page
  - **Hierarchical way to organize** documents and display them
  - **Combines semantics** (document structure) with **presentation** (document layout)
  - Allows tags to be interspersed with document content e.g., `<head>` - these are not displayed, but are directives to the layout engine
  - Also has **pointers to the JavaScript** code (e.g., `<script>`)
- Is retrieved by the browser and parsed into a tree called the Document Object Model (DOM)
  - Common way for elements to interact with the page
  - Can be read and modified by the JavaScript code
  - Modifications to the DOM are rendered by browser

HTML (HyperText Markup Language)

Example:



CSS (Cascading Style Sheets)

- CSS **separate the content** of the page **from its presentation**
- Language for **specifying how** (HTML) **documents are presented** to users (separate from content)
- **Declarative** – set of rules and their actions
  - Makes it easy to modify and maintain the website
  - An element on LHS and action to apply on RHS
  - Ensure uniformity by applying the rule to all elements of the webpage in the DOM
  - Allows different rules to be specified for different display formats (e.g., printing versus display)

CSS: Example

```
1  <html>
2      <head>
3          <title>Sample document</title>
4          <link rel="stylesheet" href="style1.css">
5      </head>
6      <body>
7          <p>
8              <strong>C</strong>ascading
9              <strong>S</strong>tyle
10             <strong>S</strong>heets
11         </p>
12     </body>
13 </html>
```

CSS: Example

```
1 strong {color: red;}
```

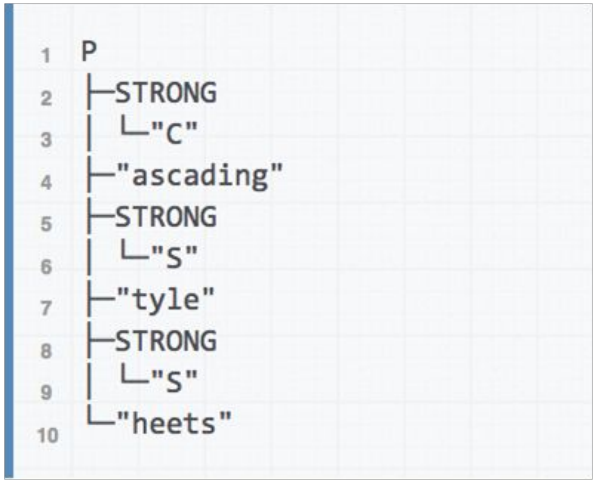
- **`strong`** → tag to match
- **`color: red;`** → attribute: value

```
1 <p>
2     <strong>C</strong>ascading
3     <strong>S</strong>tyle
4     <strong>S</strong>heets
5 </p>
```

Result: Cascading Style Sheets

CSS: How does it work?

- Apply styles to the DOM tree of the web page
- CSS rule applies to DOM nodes matching tag, and their descendants (unless overridden)



Here, all STRONG tags will be matched; all descendants of STRONG tags will be styled.

CSS: Inheritance

- All descendants of a DOM node inherit the CSS styles ascribed to it unless there is a "more-specific" CSS rule that applies to them
- Always apply style rules in top down order from the root of the DOM tree and overriding the rules as and when appropriate
    - Can be implemented with an in-order traversal

```
1 p {color:blue; text-decoration:underline}
2 strong {color:red}
```

```
1 <p>
2     <strong>C</strong>ascading
3     <strong>S</strong>tyle
4     <strong>S</strong>heets
5 </p>
```

Result:

CSS: Class and ID

- CSS rules can also apply to elements of a certain class or an element with a specific ID

```
.key {
  color: green;
}
```

```
#principal {
  font-weight: bolder;
}
```

```
<p class="key" id="principal">
```

CSS: Rules and Priority

- What to do when rules conflict with each other ?
  - Always apply the "most specific selector"
- "Most-specific" (‘>’ represents specificity):
  - Selectors with IDs > Classes > Tags
  - Direct rules get higher precedence over inherited rules (as before)

CSS: Example

```html
<!doctype html>
<html>
  <head>
  <meta charset="UTF-8">
  <title>Sample document</title>
  <link rel="stylesheet" href="style1.css">
  </head>
  <body>
    <p id="first">
      <strong class="carrot">C</strong>ascading
      <strong class="spinach">S</strong>tyle
      <strong class="spinach">S</strong>heets
    </p>
    <p id="second">
      <strong>C</strong>ascading
      <strong>S</strong>tyle
      <strong>S</strong>heets
    </p>
  </body>
</html>
```

```css
strong { color: red; }
.carrot { color: orange; }
.spinach { color: green; }
#first { font-style: italic; }
```

Cascading Style Sheets

Cascading Style Sheets

CSS: Selectors based on Relationships

- Selectors can also be based on relationships between elements in the DOM tree
  - A E : Any element E that is a descendant of A
  - A > E: Any element E that is a child of A
  - E : first-child: Any element E that is the first child of its parents
  - B + E : Any element E that is the next sibling of B element (i.e., B and E have the same parent)

CSS: Pseudo-Class Selectors

- CSS Selectors can also involve actions external to the DOM called pseudo-classes
- Visited: Whether a page was visited in the history
- Hover: Whether the user hovered over a link
- Checked: Whether a check box was checked

```
1  Selector : pseudo-class {
2      property: value
3  }
```

CSS: Example

```html
<div class="menu-bar">
  <ul>
    <li>
      <a href="example.html">Menu</a>
      <ul>
        <li>
          <a href="example.html">Link</a>
        </li>
        <li>
          <a class="menu-nav" href="example.html">Submenu</a>
          <ul>
            <li>
              <a class="menu-nav" href="example.html">Submenu</a>
              <ul>
                <li><a href="example.html">Link</a></li>
                <li><a href="example.html">Link</a></li>
                <li><a href="example.html">Link</a></li>
                <li><a href="example.html">Link</a></li>
              </ul>
            </li>
            <li><a href="example.html">Link</a></li>
          </ul>
        </li>
      </ul>
    </li>
  </ul>
</div>
```

```css
div.menu-bar ul ul {
  display: none;
}

div.menu-bar li:hover > ul {
  display: block;
}
```

The first rule says that for all ‘div’ elements of class ‘menu-bar’, in which an ul element is a descendant of another ul, do not display the second element

The second rule says that for all ‘div’ elements of class ‘menu-bar’, in which an ul element is a child of an li element, display it and the entire block, if the mouse hovers over the second element

DOM and Events

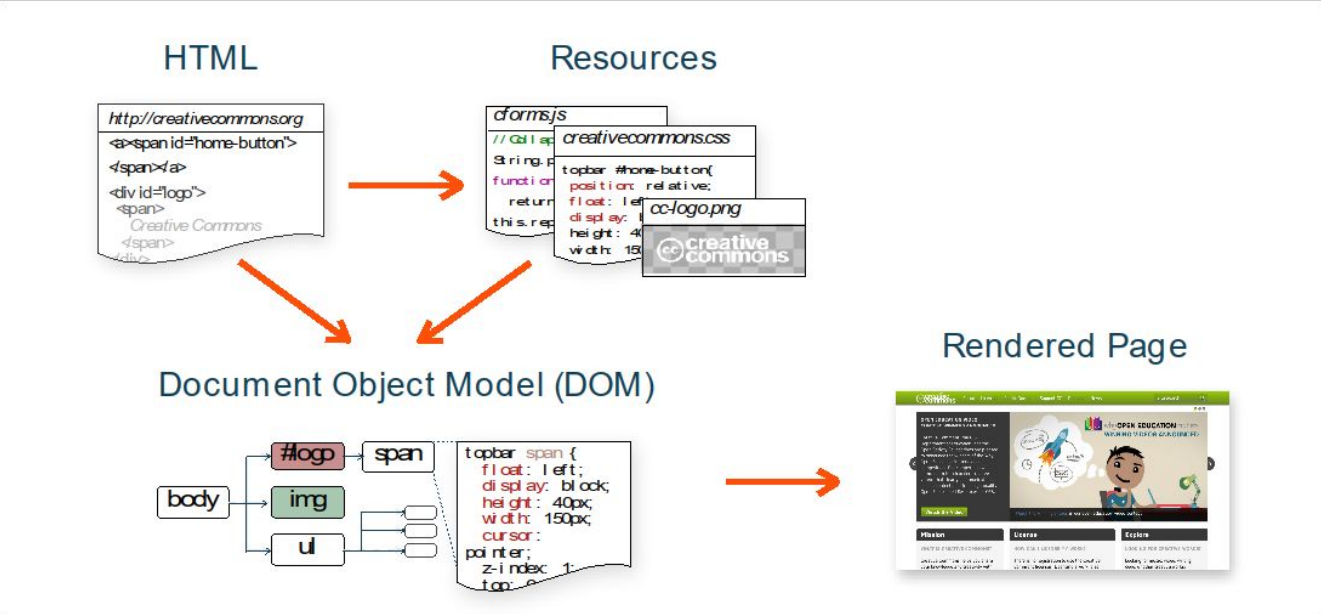
1. HTML and CSS
2. **DOM and Events**
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. Exam Logistics

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

HTML: Browser's View of HTML - DOM

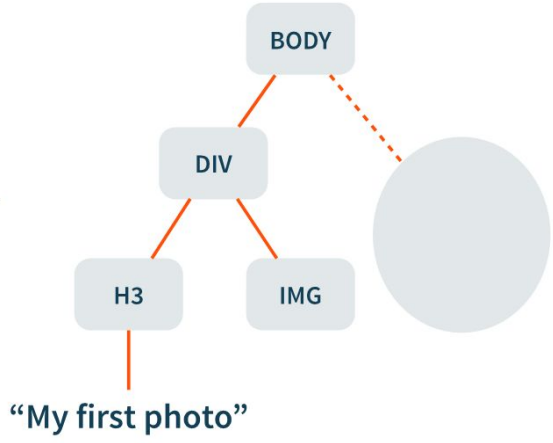
HTML is parsed by the browser into a tree structure - Document Object Model (DOM)



HTML: DOM Example

Often one-to-one correspondence between HTML and the DOM rendered by browser

```
<body>
  <div class="photo">
    <h3>My first photo</h3>
    <img src="picture1.jpg"/>
  </div>
...
</body>
```



HTML: Why is DOM important?

- Common data-structure for holding elements of a web-page (HTML, CSS, JavaScript etc.)
  - No need to worry about parsing HTML, CSS etc.
- Corresponds almost exactly to the browser's rendered view of the document
  - Changes to the DOM are made (almost) immediately to the rendered version of the webpage
  - Heavily used by JavaScript code to make changes to the webpage, and also by CSS to style the page

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

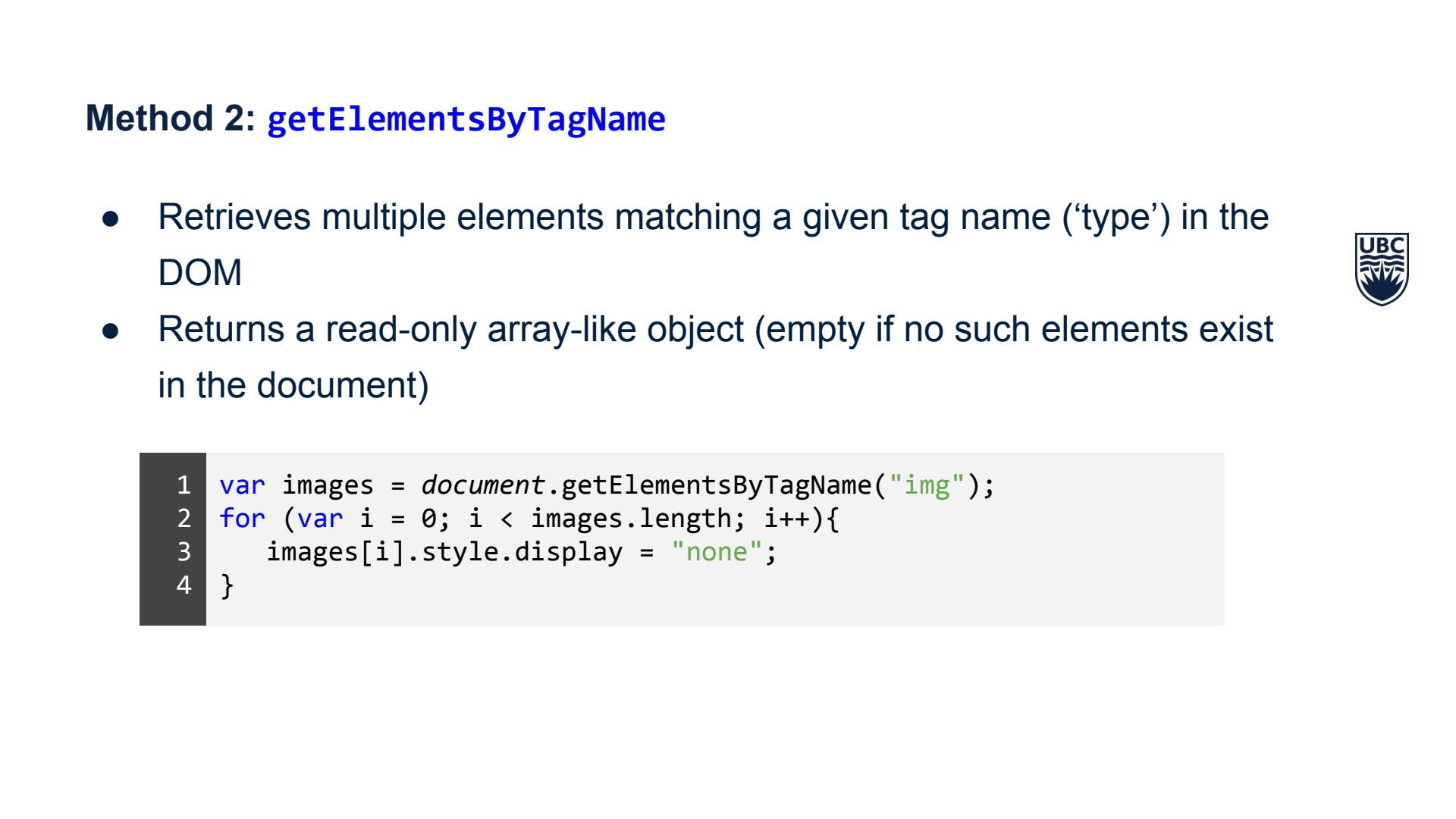
Method 1: `getElementById`

- 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

```
"#nav"          // Any element with id="nav"

"div"           // Any <div> element

".warning"      // Any element with "warning" class

"#log span"     // Any <span> descendant of id="log"

"#log > span" // Any <span> child element of id="log"

"body > h1:first-child"      // first <h1> child of <body>

"div, #log"     // All <div> elements and element with id="log"

```

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
```

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

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!");
```

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);
```

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

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

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
```

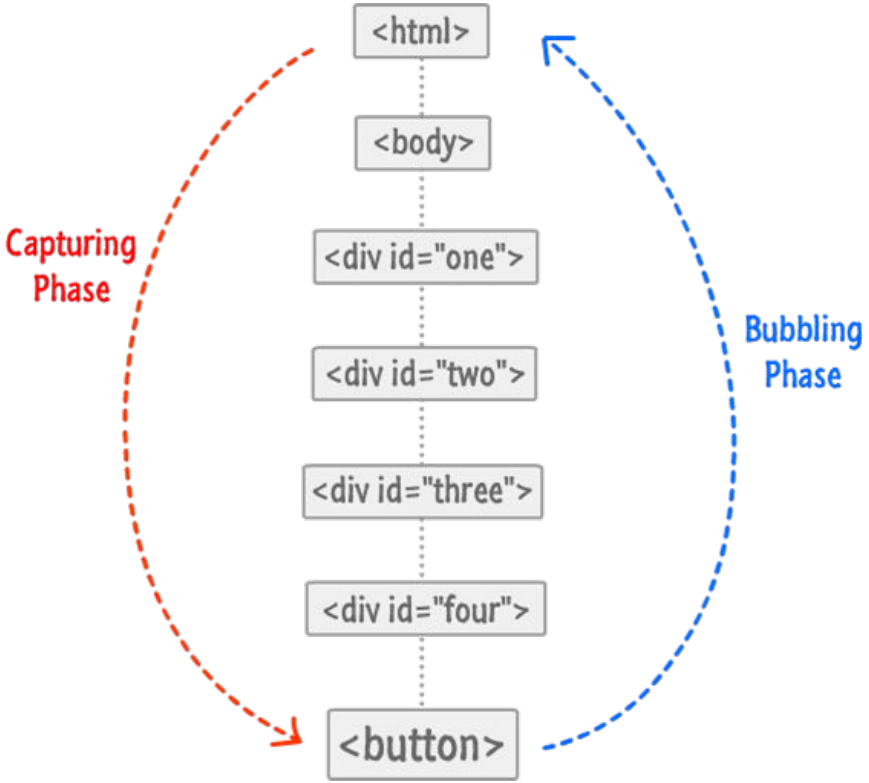
Event Handler Context

- 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

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

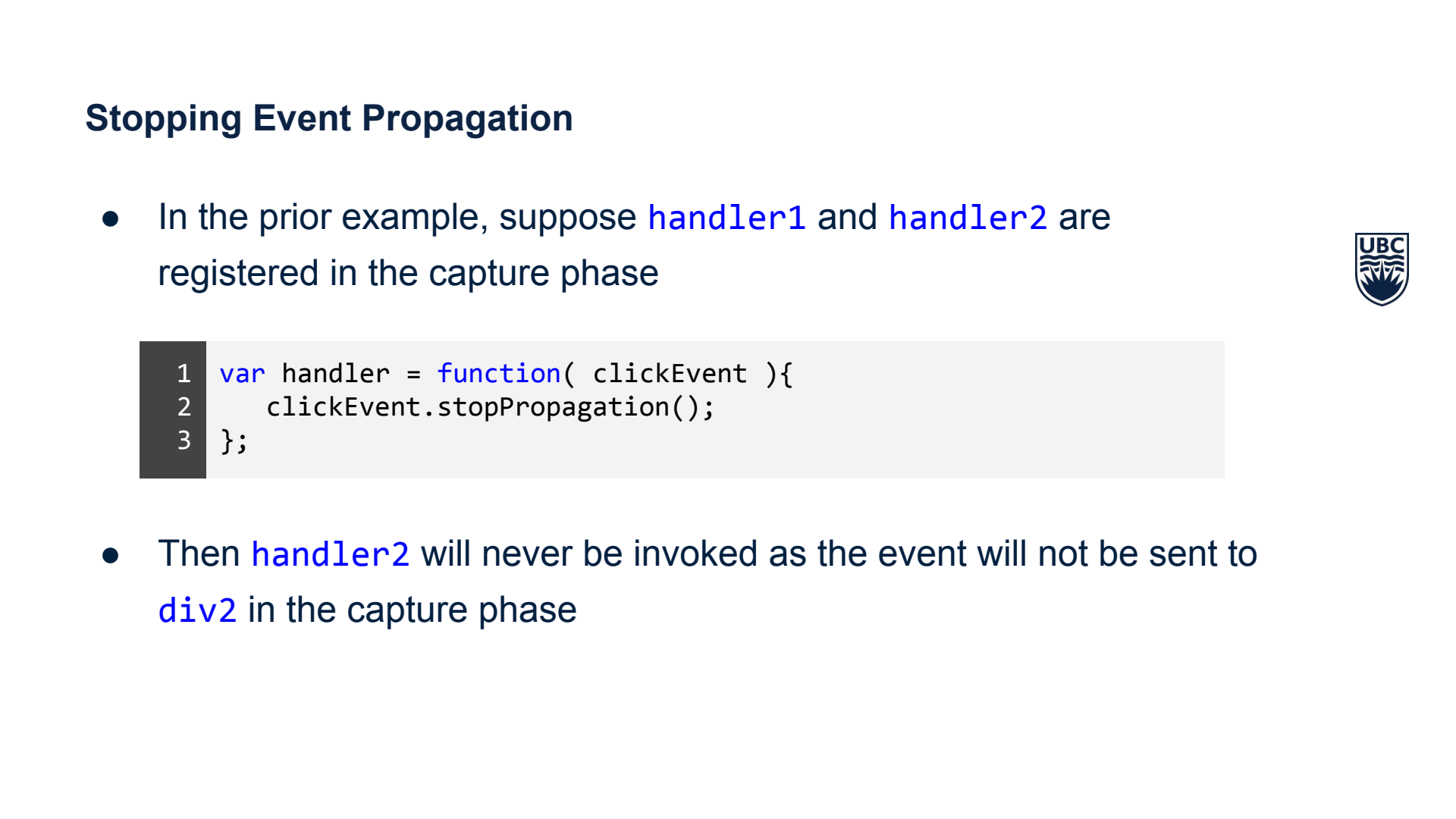
- Assume that the `<div>` element `'two'` is clicked.
- `handler1` is invoked before `handler2` as both are registered during the capture phase.

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`

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 */ }
```

JavaScript

1. HTML and CSS
2. DOM and Events
3. **JavaScript**
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. Exam Logistics

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

Introduction to JavaScript: HTML `<script>` tag

1. **Inline JavaScript** - directly as part of the HTML document


```
<html>
     <head>
          <title>My JavaScript Page</title>
     </head>
     <body>
          Hello
          <script type="text/javascript">
          var i = 2+2;
          document.writeln(i);
          </script>
          World
     </body>
</html>
```

Introduction to JavaScript: HTML `<script>` tag

2. External JavaScript - asynchronously loaded via HTTP

```
<html>
     <head>
          <title>My JavaScript Page</title>
     </head>
     <body>
          Hello
          <script type="text/javascript" src="app.js"></script>
          World
     </body>
</html>
```

Data Types: Primitive Objects

- Boolean: `true` or `false`
- Number: `1`, `3.1412`, `1.6e3`, `01011001`
  - There is no distinction between Integers and Floating Point Numbers
- String: `"Hello"`, `'World'`

```
1  b = false;
2  n = 42;
3  s = "Hello World!";
4
5  console.log(typeof b);   // prints: boolean
6  console.log(typeof n);    // prints: number
7  console.log(typeof s);    // prints: string
```

Data Types: Array

- Used to hold multiple objects in a sequence
- Arrays in JavaScript are **not typed** and **dynamic**
  - Can hold **any object** regardless of type
  - Can **add** or **remove** items **anytime**

```
1  my_list = [ "Hello World!", 42, true ];
2
3  console.log(my_list[0]);   // prints: Hello World!
4  console.log(my_list[1]);   // prints: 42
5  console.log(my_list[2]);   // prints: true
6
7  my_list.push("JavaScript is great!");
8
9  console.log(my_list[3]);   // prints: JavaScript is great!
```

Data Types: Array

- Used to hold multiple objects in a sequence
- Arrays in JavaScript are **not typed** and **dynamic**
  - Can hold **any object** regardless of type
  - Can **add** or **remove** items **anytime**
- Arrays can store Arrays

```
my_2d_list = [
   [ "A", "B", "C" ],
   [ "D", "E", "F" ],
   [ "G", "H", "I" ]
];

console.log(my_2d_list[1][2]);   // prints: F
```

Data Types: Associative Array

- key-value data structure, similar to Python dictionary

```
1  dictionary = {
2     ab: "Alberta",
3     bc: "British Columbia",
4     on: "Ontario",
5     qc: "Quebec"
6  };
7
8  console.log(dictionary.bc);     // prints: British Columbia
9
10 code = "qc";
11 console.log(dictionary[code]);  // prints: Quebec
12
13
```

Data Types: Associative Array

- Can be arbitrarily nested

```
member = {
   name: "Alice",
   age: 25,
   address: {
      province: "BC",
      city: "Vancouver",
      street: "123 Main Street"
   }
};

console.log(member.address.city);  // prints: Vancouver
member.phone = "012-345-6789";
```

Data Types: Associative Array

- Objects have **properties**
  - Properties point to other Objects in the heap
  - Properties can be dynamically added, removed, or re-assigned a value

```
1  member = {};
2
3  member.name = "Alice";
4  member.phone = "012-345-6789";
5
6  console.log(member.name);  // prints: Alice
7
8  delete member.name
9  console.log(member.name);  // prints: undefined
```

TRY IT!

Data Types: Function

- Function is also an Object

```
select_max = function (number_list){
   /* do something */
};

console.log(select_max); // prints: [Function: select_max]
                         // *output may differ between browsers
```

Data Types: Function

- Function is also an Object

```
1  select_max = function (number_list){
2     /* do something */
3  };
4
5  console.log(select_max); // prints: [Function: select_max]
6                           // *output may differ between browsers
7
8  select_max.description
9    = "Returns the maximum value from an Array of numbers";
10
11 console.log(select_max.description);
12   // prints: Returns the maximum value from an Array of numbers
13
```

Data Types: `null` and `undefined`

- `null` is actually something
  - It indicates the **absence** of a value
  - `null` itself is an object
  - Big **source of confusion**; dubbed as a major **BUG**
- `undefined` is when there is actually nothing

```
1  null_data = null;
2  undefined_data = undefined;
3
4  console.log(typeof null_data);      // prints: object
5  console.log(typeof undefined_data); // prints: undefined
6
7  console.log(window.foo); // prints: undefined
```

TRY IT!

Data Types: Summary

Primitive Types:

- boolean
- number
- string
- undefined

Complex Types:

- function
- object

Important Notes:

- null vs undefined

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);
```

Statements and Expressions: Variable Declaration

- `var` keyword used to declare variables
- No **type**s - JS is "duck-typed"

```
1  "use strict";
2
3  var width;
4  var height;
5
6  var width, height, length;
7
8  var width = 10;
9  var width = 20, height = 5, length = 10;
10 var volume = width * height * length;
11
12 console.log(volume);
```

Statements and Expressions: Assignment Statement

- = operator used to assign a new value to a reference
  - In strict mode, assignment is allowed only for declared variables

```
1  "use strict";
2
3  var width = 20, height = 5, length = 10;
4  var volume1 = width * height * length;
5
6  console.log(volume1);
7
8  width = 10;
9  height = 15;
10 var volume2 = width * height * length;
11
12 console.log(volume1, volume2);
```

Statements and Expressions: Binary/Unary Expression

Arithmetic

```
1  a + b;
2  a - b;
3  a * b;
4  a / b;
5  a % b;
6
7
8
9
10
11
12
13
```

Bitwise

```
1  ~b;
2
3  a & b;
4  a | b;
5  a ^ b;
6  a ~ b;
7  a << b;
8  a >> b;
9  a >>> b;
10
11
12
13
```

Logical

```
1  !b;
2
3  a == b;
4  a === b;
5  a != b;
6  a !== b;
7  a > b;
8  a >= b;
9  a < b;
10 a <= b;
11
12 a && b;
13 a || b;
```

Statements and Expressions: Binary/Unary Expression

- 2 Different notions of **equality**
  - `a == b` : a and b are "equivalent"
    - Loose equality
    - Equal if the values are equivalent
    - **Type coercion** performed implicitly
  - `a === b` : a and b are "identical"
    - Strict equality
    - **Type** and **value** are both equal
    - For an Object, its **value** is its location in the heap ("pointer")

Class Activity:

What would be the output of the following code?

```
1  var x = 5;
2  console.log(x == 5);       // prints?
3  console.log(x != 5);       // prints?
4  console.log(x >= 5);       // prints?
5  console.log(x < 5);        // prints?
6
7  console.log(x == "5");     // prints?
8  console.log(x === "5");    // prints?
9  console.log(x != "5");     // prints?
10 console.log(x !== "5");    // prints?
11 console.log(x !== 5);      // prints?
12
13
```

Class Activity:

What would be the output of the following code?

```
var x = 5;
console.log(x == 5);       // prints: true
console.log(x != 5);       // prints: false
console.log(x >= 5);       // prints: true
console.log(x < 5);        // prints: false

console.log(x == "5");     // prints: true
console.log(x === "5");    // prints: false
console.log(x != "5");     // prints: false
console.log(x !== "5");    // prints: true
console.log(x !== 5);      // prints: false


```

Class Activity:

What would be the output of the following code?

```
var x = { name: "Foo", value: 5 };
var a = { name: "Foo", value: 5 };
var b = x;

console.log(a.name === x.name);   // prints?
console.log(a.value === x.value); // prints?
console.log(a === x);             // prints?
console.log(b === x);             // prints?
```

Class Activity:

What would be the output of the following code?

```
var x = { name: "Foo", value: 5 };
var a = { name: "Foo", value: 5 };
var b = x;

console.log(a.name === x.name);   // prints: true
console.log(a.value === x.value); // prints: true
console.log(a === x);             // prints: false
console.log(b === x);             // prints: true
```

Statements and Expressions: Call Expression

- Function calls have the form:
  - `functionName (argument1, argument2, argument3, ...)`
  - Invokes function referred by `functionName` with the given *arguments*
  - Same as many other languages

```
1  console.log("Foo");
2  alert("Foo");
3  setTimeout(alert, 1000, "Foo");
4  setInterval(alert, 1000, "Foo");
```

Statements and Expressions: Function Declaration

- Functions can be declared with the `function` keyword
  - Can accept arbitrary arguments
  - No need to specify the return type
  - Lexical scoping - functions can have local variables that inherit the local context at the time of declaration *(we will cover this in more detail later)*


```
1  function density(mass, width, height, length){
2     var volume = width * height * length;
3     return mass / volume;
4  };
5
6  density(10, 20, 5, 10);
```

Variable and Function Declaration: Hoisting

Variable and Function Declarations are **hoisted**

- **Processed before** other expressions in the program
- To avoid confusion, **best to put** Variable Declarations and Function Declarations **at the top** of the program

```
1  console.log(density); // prints: [Function: density]
2
3  function density(mass, width, height, length){
4     var volume = width * height * length;
5     return mass / volume;
6  };
```

Function: Recap

- JavaScript functions are **not typed**
- JavaScript functions are **first-class objects**
  - They can be **assigned to variables**
  - They can be **passed as arguments** into another function call
  - Functions can **return other functions**
- Function Declarations have the format:
  - `function functionName (arg, arg, ...) { /* body */ }`
- Function Expressions can create **anonymous functions**
  - `var x = function (arg, arg, ...) { /* body */ }`

Function: Variadic Function

- JavaScript functions **cannot be overloaded**
- To emulate function overloading, we can define a **variadic function** using the special `arguments` object

```
1  function sayHi (){
2     if (arguments.length < 3)
3        console.log("Hi " + arguments[0] + " " + arguments[1]);
4     else
5        console.log("Hi " + arguments[0] + " " + arguments[1] + " " +
6  arguments[2]);
7  };
8  sayHi("Alice", "Brown");  // prints: Alice Brown
```

TRY IT!

Function: Immediate Evaluation

- Function Expressions can be evaluated **immediately after definition**
  - Useful for capturing dynamic variables when creating a closure (coming up later)

```
1  var y = function foo (x){
2     return x + 10;
3  };
4
5  console.log(y);     // prints: [Function: foo]
```

TRY IT!

Function: Immediate Evaluation

- Function Expressions can be evaluated **immediately after definition**
  - Useful for capturing dynamic variables when creating a closure (coming up later)

```
1  var y = (function foo (x){
2     return x + 10;
3  })(1);
4
5  console.log(y);     // prints: 11
```

TRY IT!

Function: Nesting

- JavaScript functions can be nested arbitrarily

```
1  function alpha (x){
2     var i = x + x;
3     function bravo (y){
4        var j = y + i;
5        function charlie (z){
6           var k = z + j;
7           return k;
8        }
9        return charlie(j);
10    }
11    return bravo(i);
12 };
13
14 console.log(alpha(1));  // prints?
```

TRY IT!

Function: Nesting

- JavaScript functions can be nested arbitrarily

```
1  function alpha (x){
2     var i = x + x;
3     function bravo (y){
4        var j = y + i;
5        function charlie (z){
6           var k = z + j;
7           return k;
8        }
9        return charlie(j);
10    }
11    return bravo(i);
12 };
13
14 console.log(alpha(1));  // prints: 8
```

TRY IT!

Function: Scope

- Each function creates its own **scope** when invoked

```
1  function alpha (x){
2     var i = x + x;
3     function bravo (y){
4        var i = y + y;
5        console.log(i);    // prints: 4
6     }
7     bravo(i);
8     console.log(i);       // prints: 2
9  };
10
11 alpha(1);
12
13
14
```

TRY IT!

Function: Scope

- A child function has access to its parent's **scope**

```
1  function alpha (x){
2     var i = x + x;
3     function bravo (y){
4        i = y + y;
5        console.log(i);    // prints: 4
6     }
7     bravo(i);
8     console.log(i);       // prints: 4
9  };
10
11 alpha(1);
12
13
14
```

TRY IT!

Function: Scope

- A parent function does not have access to its child's **scope**

```
1  function alpha (x){
2     var i = x + x;
3     function bravo (y){
4        var j = y + y;
5        console.log(i);    // prints: 2
6     }
7     bravo(i);
8     console.log(j);       // throws: ReferenceError: j is not defined
9  };
10
11 alpha(1);
12
13
14
```

TRY IT!

Function: First-Class Objects

- Functions can be passed to other functions as arguments

```
1  function filter (list, f){
2     var arr = [];
3     for (var i = 0; i < list.length; i++){
4        if (f(list[i]) === true) arr.push(list[i]);
5     }
6     return arr;
7  };
8
9  var myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
10 var filtered = filter(myList, function (item){
11    return (item < 5);
12 });
13 console.log(filtered);     // prints: 0, 1, 2, 3, 4
14
```

TRY IT!

JavaScript: Callback and Closure

1. HTML and CSS
2. DOM and Events
3. JavaScript
    a. **Callback and Closure**
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. Exam Logistics

Callback Function

- Callback functions are just regular functions, used in a certain way
  - They are not some special function type
- Used for performing **asynchronous operations**
  - JavaScript applications are **full of asynchronous operations**, so callbacks are used very frequently
  - Most notable examples are **event listeners**
- Why use callbacks?
  - Some operations are **fundamentally asynchronous** (e.g., network requests)
  - We **don’t want to wait for result indefinitely**. We would rather get a **call back** when something is done.

Callback Function

```
1  function asyncFunction (arg1, arg2, callback){
2     /*
3        do some asynchronous operations
4      */
5        callback(result);    // invoke callback when result is available
6     return null             // return immediately
7  };
8
9  asyncFunction(val1, val2, function(result){
10    /* do something with result */
11 });                        // this call returns null immediately
12
13 /* do other things */
14
```

Closure Function

- Closure functions are just regular functions, used in a certain way
  - They are not some special function type
- Closures are functions that carry references outside of their own scope
  - Used to hide objects while still providing the functionality
  - Used to create stateful functions

Closure Function

```
1  function makeCounter (initial, increment){
2     var count = initial;
3     return function next(){
4        count += increment;
5        return count;
6     }
7  };
8  var counter1 = makeCounter(3, 1);
9  var counter2 = makeCounter(5, 5);
10 console.log(counter1());          // prints: 4
11 console.log(counter2());          // prints: 10
12 console.log(counter1());          // prints: 5
13 console.log(counter2());          // prints: 15
```

TRY IT!

Closure Function

```
1  function makeCounters (n){
2     var counts = [];
3     var counters = [];
4     for (var i = 0; i < n; i++){
5        counts[i] = 0;
6        counters[i] = function next(){
7           counts[i] ++;
8           return counts[i];
9        };
10    }
11    return counters;
12 };
13
14 var cs = makeCounters(10);
15 console.log( cs[0]() );      // prints?
16 console.log( cs[4]() );      // prints?
```

Closure Function

```
1  function makeCounters (n){
2     var counts = [];
3     var counters = [];
4     for (var i = 0; i < n; i++){
5        counts[i] = 0;
6        counters[i] = function next(){
7           counts[i] ++;
8           return counts[i];
9        };
10    }
11    return counters;
12 };
13
14 var cs = makeCounters(10);
15 console.log( cs[0]() );      // prints: NaN
16 console.log( cs[4]() );      // prints: NaN
```

Closure Function

```
1  function makeCounters (n){
2     var counts = [];
3     var counters = [];
4     for (var i = 0; i < n; i++){
5        counts[i] = 0;
6        counters[i] = (function (j){
7           return function next(){
8              counts[j] ++;
9              return counts[j];
10          };
11       })(i);
12    }
13    return counters;
14 };
15
16 var cs = makeCounters(10);
```

JavaScript: ECMAScript 2015 (ES6)

1. HTML and CSS
2. DOM and Events
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. Exam Logistics

Object-oriented Programming

`this` keyword

- `this` refers to the object on which the function is called

```
1  function accelerate (fuel){
2    this.velocity += fuel * this.power;
3  }
4  var myCar = {
5    name: "Smart",
6    power: 1,
7    velocity: 0,
8    accelerate: accelerate
9  }
10 myCar.accelerate(10);
11 accelerate(12);         // What is "this"?
```

Object-oriented Programming

`this` keyword

- Function objects have a method called `bind`, which can be used to "lock" what `this` refers to

```
1  function accelerate (fuel){
2    this.velocity += fuel * this.power;
3  }
4  var myCar = {
5    name: "Smart",
6    power: 1,
7    velocity: 0,
8    accelerate: accelerate
9  }
10 myCar.accelerate(10);
11 accelerate.bind(myCar)(12);        // What is "this"?
```

Object-oriented Programming

new keyword

```
1  function Car (name, power=1){
2     this.name = name;
3     this.power = power;
4     this.velocity = 0;
5  };
6  Car.prototype = {};
7  Car.prototype.accelerate = function(fuel){
8     this.velocity += fuel * this.power;
9  };
10
11 var myCar = new Car("Smart");
12 myCar.accelerate(10);
13
14
```

Object-oriented Programming

`new` keyword

- Invoking `new Foo(arg1, arg2)` will perform the following:
  - Create a new object by shallow-copying `Foo.prototype`; we will refer to this new object as `newFoo` in this slide
    - Since this is a "shallow copy", functions and objects bound to `newFoo`'s properties will all point to the corresponding objects on `Foo.prototype`. (e.g., `myCar.accelerate === Car.prototype.accelerate`)
  - Invoke the `Foo` function in the context of the newly created object `newFoo` (i.e., `Foo.bind(newFoo)(arg1, arg2)`); we thus refer to the `Foo` function as "a constructor"

Object-oriented Programming

`class` and `constructor` keyword

```
1  class Car {
2    constructor (name, power=1){
3      this.name = name;
4      this.power = power;
5      this.velocity = 0;
6    }
7    accelerate (fuel){
8      this.velocity
9        += fuel * this.power;
10   }
11 }
12
13 var myCar = new Car("Smart");
14 myCar.accelerate(10);
```

Object-oriented Programming

`extends` and `super` keyword

```
1  class RacingCar extends Car {
2    constructor (name){
3      super(name, 3.5);
4    }
5
6    turbo (fuel){
7      this.velocity += fuel * this.power * 1.5;
8    }
9
10 }
11
12 var superCar = new RacingCar("F1");
13 superCar.accelerate(10);
14 superCar.turbo(5);
```

Functional Programming

- JavaScript supports functional programming
- When used appropriately, `function`s can implement pure functions
  - Except it is not actually a pure function
  - Keywords like `this`, `arguments` make JavaScript functions impure
- ES6 introduces **arrow functions** to support real functional programming

Functional Programming

- Arrow functions are **not replacements** for ES5 functions
- Arrow functions are **anonymous functions**
- `this` and `arguments` inside arrow functions are lexically bound

Syntax Example:

```
1  (radius, height) => {
2     return radius * radius * Math.PI * height;
3  }
4
5  (radius, height) => (radius * radius * Math.PI * height);
```

Functional Programming

- Pure functions
  - Always returns the same value given the same arguments
  - Have no side effects like mutating an external object (e.g., I/O, network resource, variables outside of its scope)
  - Examples:
    - area of circle, distance between 2 points in 3-dimensional space
- Impure functions
  - Might depend on an external context
  - Might change an external object
  - Examples:
    - `Date.now()`
    - `console.log()`

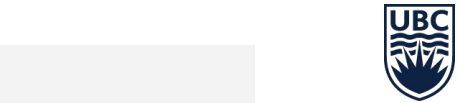
Functional Programming

Arrow function syntax

```
1  // Regular function
2  function(arg1, arg2){
3     // do some stuff here
4     return arg1 + arg2;
5  }
6
7  // Imperative usage
8  (arg1, arg2) => {
9     // do some stuff here
10    return arg1 + arg2;
11 }
12
13 // Pure function
14 (arg1, arg2) => (arg1 + arg2);
```

Functional Programming

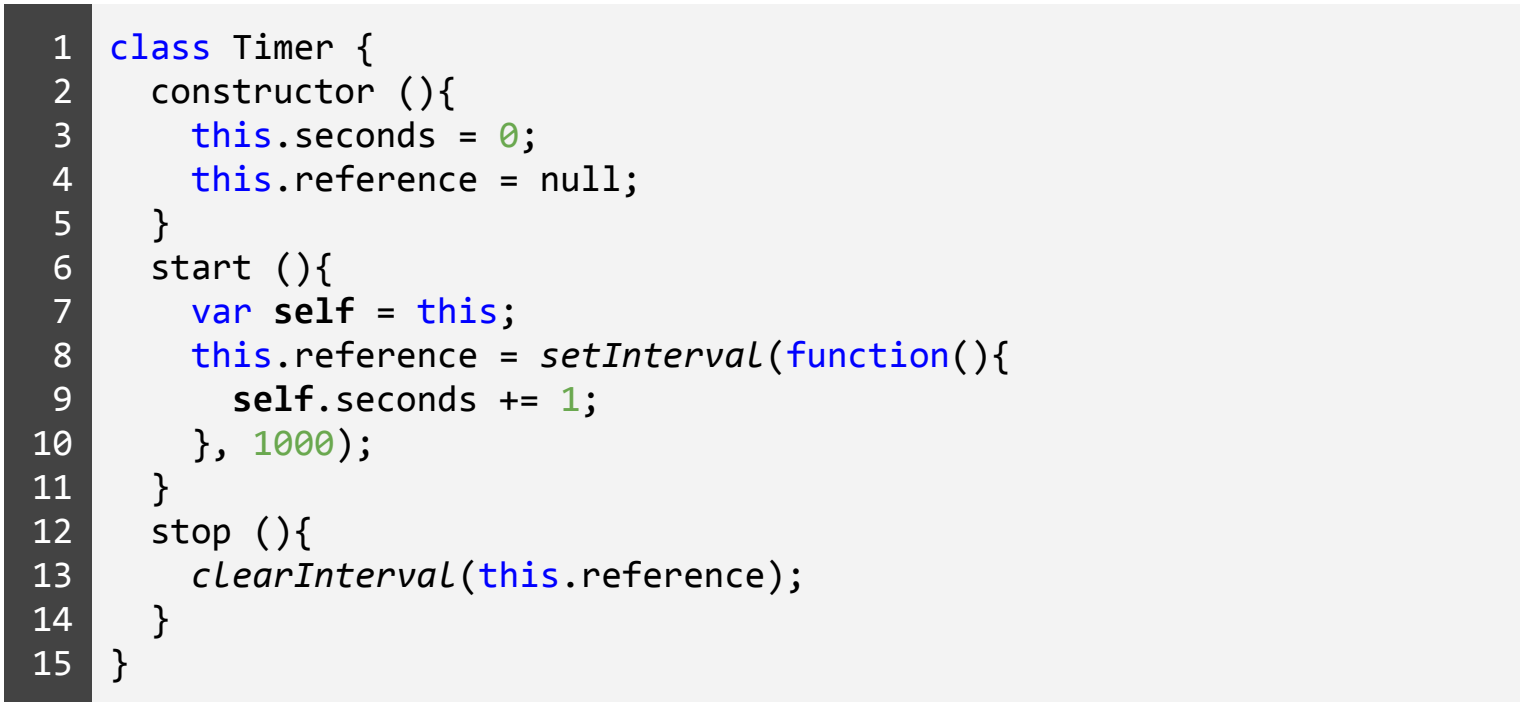
- Arrow Function usage scenario



```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    this.reference = setInterval(function(){
      this.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

Functional Programming

- Arrow Function usage scenario



```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    var self = this;
    this.reference = setInterval(function(){
      self.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

Functional Programming

- Arrow Function usage scenario


```
class Timer {
  constructor (){
    this.seconds = 0;
    this.reference = null;
  }
  start (){
    this.reference = setInterval(()=> {
      this.seconds += 1;
    }, 1000);
  }
  stop (){
    clearInterval(this.reference);
  }
}
```

What is a Promise

- Promise is a new built-in object **introduced in ES6**
- Provides a **cleaner interface** for handling **asynchronous operations**
- When multiple asynchronous operations need to be made, the **callback** pattern **becomes hard to follow**
  - Scope of variables in multiple nested closures
  - Error handling for each of the callback steps

Promise

- `Promise` is an object with the following methods
  - `then (onResolve, onReject)`: used to register resolve and reject callbacks
  - `catch (onReject)`: used to register reject callback
  - `finally (onComplete)`: used to register settlement callback
- `Promise` will be in one of the three states: pending, resolved, rejected
- `Promise` also has static methods
  - `resolve (value)`: returns a `Promise` that resolves immediately to `value`
  - `reject (error)`: returns a `Promise` that rejects immediately to `error`
  - `all (promises)`: returns a `Promise` that resolves when all promises resolve
  - `race (promises)`: returns a `Promise` that resolves if any of the promises resolve

Promise

- Creating a `Promise` object
  - `new Promise(func)`: The `Promise` constructor expects a single argument *`func`*, which is a function with 2 arguments: `resolve`, `reject`
  - `resolve` and `reject` are callback functions for emitting the result of the operation
    - `resolve(result)` to emit the result of a successful operation
    - `reject(error)` to emit the error from a failed operation

```
1  var action = new Promise((resolve, reject)=> {
2     setTimeout(()=> {
3        if (Math.random() > 0.5) resolve("Success!");
4        else reject(new Error("LowValueError"));
5     }, 1000);
6  });
7
```

Promise

- Using the result of a `Promise` fulfillment through the `then` method
  - `then(onResolve, onReject)`: used to register callbacks for handling the result of the `Promise`. It returns another `Promise`, making this function **chainable**
  - `onResolve` is called **if the previous `Promise` resolves**; it receives the resolved value as the only argument
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
1  action.then(
2     (result)=> console.log(result), // result: "Success!"
3     (error)=> console.log(error)    // error: Error("LowValueError")
4  )
5  .then(()=> console.log("A"))
6  .then(()=> console.log("B"));
```

Promise

- The `catch` method is used to handle the result of a rejected `Promise`
  - `catch(onReject)`: used to register a callback for handling the result of the failed `Promise`. It returns another `Promise`, making this function **chainable**
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err));

```

Promise

- The `finally` method is used to register a callback to be called when a `Promise` is settled, regardless of the result
  - `finally(onComplete)`: It returns another `Promise`, making this function **chainable**
  - `onComplete` is called **if the previous `Promise` is settled**


```
1  action.then(
2     (result)=> console.log(result), // result: "Success!"
3     (error)=> console.log(error)    // error: Error("LowValueError")
4  )
5  .catch((err)=> console.log(err))
6  .finally(()=> console.log("The End!"));
```

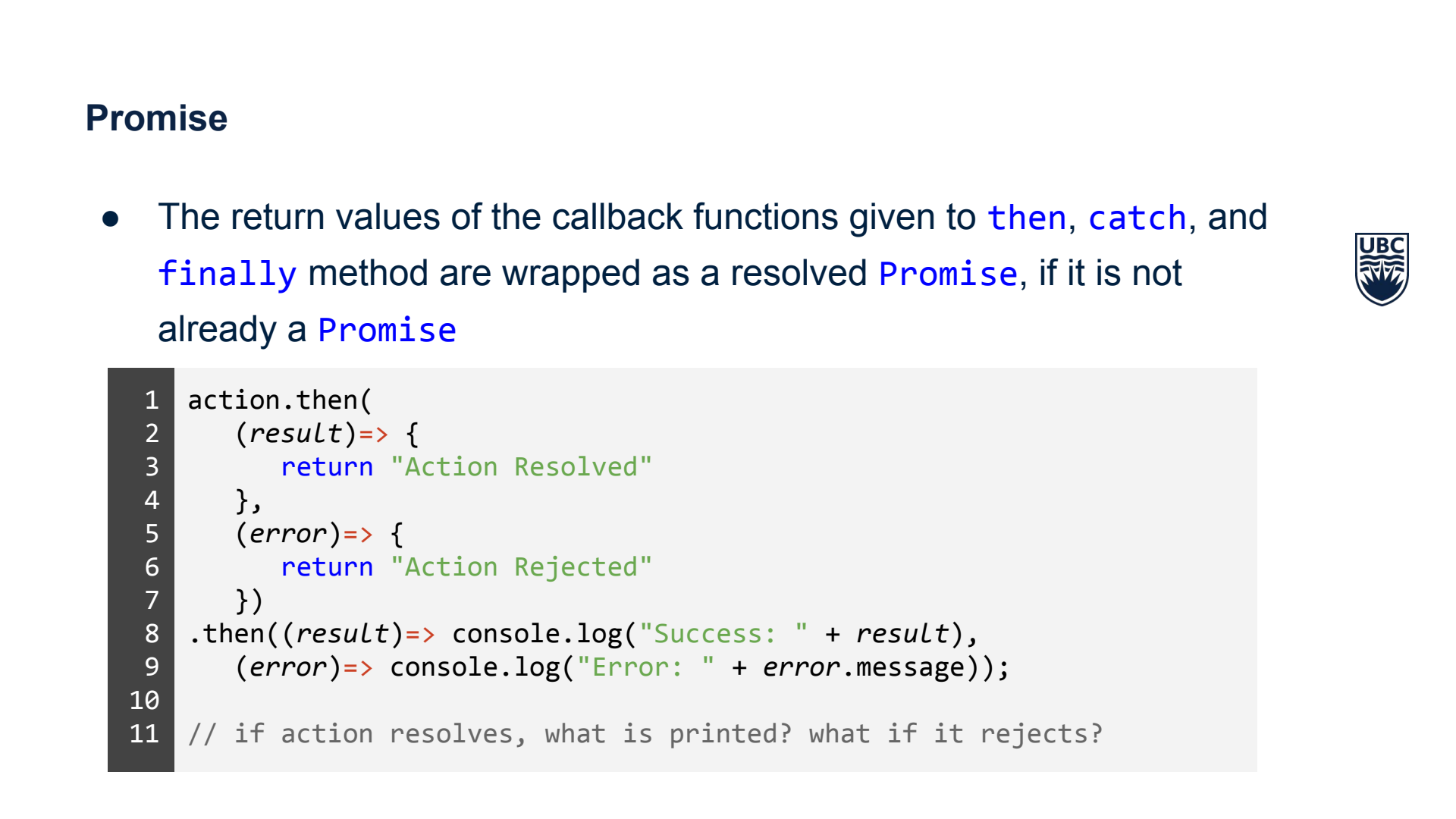
Promise

- The static functions `Promise.resolve` and `Promise.reject` are used to create a `Promise` object that immediately resolves or rejects with the given data
    - Useful when the next asynchronous operation expects a `Promise` object


```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err))
.finally(()=> console.log("The End!"));
```

Promise

- The return values of the callback functions given to `then`, `catch`, and `finally` method are wrapped as a resolved `Promise`, if it is not already a `Promise`



```
action.then(
   (result)=> {
      return "Action Resolved"
   },
   (error)=> {
      return "Action Rejected"
   })
.then((result)=> console.log("Success: " + result),
   (error)=> console.log("Error: " + error.message));

// if action resolves, what is printed? what if it rejects?
```

Promise

- Using the static function `Promise.all`, we can wait for multiple concurrent `Promise`s to be resolved (sort of like joining threads)
  - `Promise.all` accepts an Array of promises and returns a `Promise` that resolves to an array of results (in the same order as the promises given)

```
1  var multi = Promise.all([
2     new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
3     new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
4     new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
5  ]);
6
7  multi.then(
8     (results)=> console.log(results),
9     (error)=> console.log(error));
10
```

Promise

- Using the static function `Promise.race`, we can retrieve the first `Promise` to resolve out of a set of concurrent `Promises`
    - `Promise.race` accepts an Array of promises and returns the first `Promise` that resolves

```
1  var multi = Promise.race([
2     new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
3     new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
4     new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
5  ]);
6
7  multi.then(
8     (result)=> console.log(result),
9     (error)=> console.log(error));
10
```

AJAX and Node.js

1. HTML and CSS
2. DOM and Events
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. **AJAX and Node.js**
5. Session, Cookie, and Web Security
6. Exam Logistics

What is AJAX?

- Stands for "**Asynchronous JavaScript and XML**", but does not necessarily involve XML
- Mechanism for modern web applications to communicate with the server after page load
  - Without refreshing the current page
  - Request can be sent asynchronously without holding up the main JavaScript thread
- Complement of COMET (Server Push)

What is AJAX?: Usage

- Interactivity
  - To enable content to be brought in from the server in response to user requests
- Performance
  - Load the most critical portions of a web-page first, and then load the rest asynchronously
- Security (this is doubtful)
  - Bring in only the code/data that is needed on demand to reduce the attack surface of the web application

XMLHttpRequest: Creating a Request

- `XMLHttpRequest`: Constructor function for supporting AJAX
- `open`: opens a new connection to the server using the specified method (GET or POST) and to the specified URL or resource


```
1  var req = new XMLHttpRequest();
2  req.open("GET", "/example.txt");
3
4
5
6
7
8
9
10
```

XMLHttpRequest: HTTP Methods

- Two popular methods to send HTTP Request
- GET: used to retrieve data from server by client (typically with no side effects), and does not send any additional data to the server
- POST: used to store data from HTML forms on the server (typically with side effects), and sends the form data to the server

XMLHttpRequest: Sending the Request

- `send`: sends the data to the server asynchronously and returns immediately. Takes a single parameter for the data to be sent (can be omitted for GET)


```
1  var req = new XMLHttpRequest();
2  req.open("GET", "/example.txt");
3  req.send(null);    // or simply - req.send();
4  // Returns here right after the send is complete
5
6
7
8
9
10
```

XMLHttpRequest: Registering Callbacks

- Because the `send` returns right away, the data may not be sent yet (as it's sent asynchronously). Also, we have no way of knowing when the server has responded.
- We need to setup a callback to handle the various events that can occur after a send as the `onreadystatechange` function

```
1  var req = new XMLHttpRequest();
2  req.open("GET", "/example.txt");
3  req.onreadystatechange = function() {
4     // triggered whenever ready state changes
5  }
6  req.send(null);    // or simply - req.send();
7  // returns here right after the send is complete
8
```

XMLHttpRequest: Registering Callbacks

- XMLHttpRequest Status Codes
  - UNSENT (0): open has not been called yet
  - OPENED (1): open has been called
  - HEADERS_RECEIVED(2): Headers have been received
  - LOADING(3): Response is being received
  - DONE(4) : Response is done
- Don’t use the direct numerical values in code

XMLHttpRequest 1: Old Method (deprecated)

- Check whether the request's state has changed to DONE
- Check if the status of the request is 200 (denotes success in the HTTP protocol)
- Check if response is of a specific type by examining the header
- If all three conditions match, then perform the action on message receipt (e.g., parse it)


```
1  req.onreadystatechange = function() {
2     if (x.readyState == 4 && x.status == 200){
3        // do something with req.responseText
4     }
5  }
```

XMLHttpRequest 2: New Method (recommended)

- Does away with the `onreadystatechange`
  - Triggers different events depending on response
  - Much cleaner but not all browsers support it (yet)
- Events triggered by the XHR2 Model
  - Load: Response was received (does not mean that it was error-free, so still need to check status)
  - Timeout: Request timed out
  - Abort: Request was aborted
  - Error: Some other error occurred

```
1  req.onload = function() {
2     if (req.status == 200){
3        // do something with req.responseText
4     }
5  }
```

Aborting Requests

- A request can be aborted after it is sent by calling the abort method on the request
- Request may have been already sent. If so, the response is discarded
- Triggers the Abort event handler of the request


```
1  req.onabort = function() {
2     console.log("Request aborted!");
3  }
4
5
```

Timeouts

- Can also specify timeouts in the request (though this is not supported by all browsers)
- Set timeout property in ms

```
1  req.timeout = 200;    // 200 ms timeout
2  req.ontimeout = function() {
3     console.log("Request timed out");
4  }
5
```

Errors

- These occur when there is a network level error (e.g., server is unreachable).
- Trigger the error event on the request
- NOT a substitute for checking status codes

```
1  req.onerror = function() {
2     console.log("Error occurred on request");
3  }
4
5
```

Server-side JS: Advantages

- Same language for both client and server
  - Eases software maintenance tasks
  - Eases movement of code from server to client
- Much easier to exchange data between client and server, and between server and NoSQL DBs
  - Native support for JSON objects in both
- Much more scalable than traditional solutions
  - Due to use of asynchronous methods everywhere

Comparison with Traditional Solutions

- Traditional solutions on the server tend to spawn a new thread for each client request
  - Leads to proliferation of threads
  - No control over thread scheduling
  - Overhead of thread creation and context switches
- Server-side JS: Single-threaded nature of JS makes it easy to write code
  - Scalability achieved by asynchronous calls
  - Composition with libraries is straightforward

Node.js Example

```
1  console.log("Hello");   // console object is available in Node.js
2  setTimeout(() => {      // setTimeout is available
3     console.log("World");
4  }, 1000);
5
6  // Available in Node.js (can't do this in client-side JS)
7  var fs = require("fs"); // require is a Node.js built-in function
8  var content = fs.readFileSync("hello.txt");
9  console.log(content);
10
```

Node.js

- In Node.js, you use **modules to package functionality** together
- Use the `module.exports` built-in object to export a function or object as part of a module
- Use the `require` built-in function to import a module and its associated functions or objects

Exporting Functions

- Can be used to create one's own modules

```
// Calculator.js
function sum(a, b){
   return a + b;
};

// module is a special built-in object in Node.js
// module.exports object can be used to expose an API
module.exports.sum = sum;
```

Exporting Objects

- Can also export entire objects through the `module.exports`


```
1  // Point.js
2  var Point = function(x, y){
3     this.x = x;
4     this.y = y;
5  };
6
7  // module.exports is initially equivalent to {}.
8  // We replace the entire object with the Point function
9  module.exports = Point;
10
```

Using modules: `require`

- Used to express dependency on a certain module's functionality

```
1  // Imports the Calculator module
2  var calculator = require("Calculator.js");
3  calculator.sum(10, 20);
4
5  // Imports the Point module
6  var Point = require("Point.js");
7  var p = new Point(1, 2);
8
9
10
```

Points to Note

- Need to provide the **full path of the module** to the `require` function
- Need to check the return value of `require`. If it's undefined, then the module was not found.
- Only functions/objects that are exported using `export` are visible in the line that calls `require`

Event Streams

- Node.js code can define events and monitor for the occurrence of events on a stream (e.g., network connection, file etc).
- Associate callback functions to events using the `on()` or `addListener()` functions
- Trigger by calling the `emit` function

Event

- Refer to specific points in the execution
  - Example: `exit`, before a node process exists
  - Example: `data`, when data is available on connection
  - Example: `end` when a connection is closed
- Can be defined by the application and event registers can be added on streams
- Event can be triggered by the streams

Event

```
1  // import the EventEmitter constructor from built-in events library
2  var EventEmitter = require("events").EventEmitter;
3
4  // create an EventEmitter object
5  var myEmitter = new EventEmitter();
6  var onConnection = function(id){ /* ... some code */ };
7  var onMessage = function(msg){ /* ... some code */ };
8
9  // attach event listeners
10 myEmitter.on("connection", onConnection);
11 myEmitter.on("message", onMessage);
12
13 // emit events (somewhere else in the code)
14 myEmitter.emit("connection", 100);
15 myEmitter.emit("message", "hello");
```

File handling in Node

- Node.js supports two ways to read/write files
  - Asynchronous reads and writes
  - Synchronous reads and writes
- The asynchronous methods require callback functions to be specified and are more scalable
- Synchronous is similar to regular reads and writes in other languages

Synchronous Reads and Writes

- `readFileSync` and `writeFileSync` to read/write files synchronously (operations block JS)
- Not suitable for reading/writing large files
    - Can lead to large performance delays

```
1  // synchronous file read
2  var f = fs.readFileSync("sample.txt");
3
4  // synchronous file write
5  var f = fs.writeFileSync("sample.txt", "Hello World!");
6
7
8
9
10
```

Asynchronous Reads and Writes

```
1  var fs = require("fs");       // import built-in fs library
2  var length = 0;               // for keeping track of # characters
3  var fileName = "sample.txt";
4
5  // asynchronous file read takes in a callback
6  fs.readFile(fileName, function(err, buf){
7     if (err) throw err;
8     length = buf.length;
9     console.log("# of Characters = " + length);
10 })
```

Asynchronous Reads using Streams

- It's also possible to start processing a file as and when it is being read. We need to read files as event streams: `fs.createReadStream`
- Three types of events on files
  - `data`: There's data available to be read
  - `end`: The end of the file was reached
  - `error`: There was an error in reading the data

Example of Using Streams

```
1  var fs = require("fs");
2  var length = 0;
3  var fileName = "sample.txt";
4  var readStream = fs.createReadStream(fileName);
5
6  readStream.on("data", function(blob){
7     console.log("Read = " + blob.length);
8     length += blob.length;
9  });
10
11 readStream.on("end", function(){
12    console.log("Total # of Characters = " + length);
13 });
14
15 readStream.on("error", function(err){
16    console.log("Error occurred trying to read " + fileName);
17 });
```

Asynchronous Writes

- Like reads, writes can also be asynchronous. Just call `fs.writeFile` with the callback function

```
fs.writeFile("example.txt", "Hello World", function(err){
   if (err)
      console.log("Error writing to example.txt");
   else
      console.log("Finished writing data");
});
```

Writable Stream

- Like readStreams, we can define writeStreams and write data to them in blobs
  - Same events as before
  - Useful when combined with `readableStream`s to avoid buffering in memory
  - Need to call `end()` when the writing is completed

Example: Copying one file to another

```
1  var fs = require("fs");
2  var readStream = fs.createReadStream("example.txt");
3  var writeStream = fs.createWriteStream("example-copy.txt");
4
5  readStream.on("data", function(blob){
6     console.log("# of Characters = " + blob.length);
7     writeStream.write(blob);
8  });
9
10 readStream.on("end", function(){
      console.log("End of stream");
      writeStream.end();
   });
```

Alternate method: Using Pipe

```
1  var fs = require("fs");
2  var readStream = fs.createReadStream("example.txt");
3  var writeStream = fs.createWriteStream("example-copy.txt");
4
5  readStream.pipe(writeStream);
6
7
8
9
10
```

Session, Cookie, and Web Security

1. HTML and CSS
2. DOM and Events
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. **Session, Cookie, and Web Security**
6. Exam Logistics

Session: What is it?

- At a high-level, a session is something that **keeps track of the series of interactions** between communicating parties
  - It is a shared "context"
- In the context of **web applications**, a session keeps track of the communication **between the server and the client**

Session: Why is it relevant to Web Applications?

- HTTP is stateless
  - One request-response pair has no information about another request-response pair
  - Server cannot tell if 2 requests came from the same browser → server cannot maintain stateful information about the client (e.g., how many times a client viewed a page)
- **Interaction** between 2 communicating parties (client & server) involving multiple messages **require** some **state to be maintained**

Cookie: What is it?

- Cookie is a piece of data that is always passed between the server and the client in consecutive HTTP messages
- At the minimum, a cookie can store a session ID to relate multiple HTTP requests and responses
- Mainly used for:
  - Session management
  - Personalization
  - Tracking User Behaviour

Cookie: Format

- Name: indicates the type of information
- Value: the data representing the information
- Attributes: set by server only
  - Domain: specifies the scope of the cookie
  - Path: which path the cookie is allowed to be sent to
  - Expires: when the cookie should expire
  - Max-Age: the maximum age for the cookie
  - Secure: enforce cookie to be sent only via https
  - HttpOnly: do not expose the cookie to application layer (e.g., JavaScript)

Cookie: Format

- Example: Server Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: sessionid=abcdef12345
Set-Cookie: theme=default
Set-Cookie: language=en
Set-Cookie: currency=cad

<html>Hello World</html>
```

Cookie: Format

- Example: Client Request

```
GET /hello.html HTTP/1.1
Host: example.com
Cookie: sessionid=abcdef12345
Cookie: theme=default
Cookie: language=en
Cookie: currency=cad
```

Web Security: Same-Origin Policy

- Same-Origin Policy says only scripts loaded from the same origin can be executed in the page
  - Enforced by all browsers
- Intent: Two different web domains should not be able to tamper with each other's contents
- Easy to state, but many exceptions in practice
  - Visual display is shared
  - Timing and DOM events are shared
  - Cookies can be shared
  - Send/receive messages for Cross-Origin Requests

Web Security: Same-Origin Policy

- Assign an origin for each resource in a web page (e.g., cookies, DOM sub-tree, network)
  - A script can only access elements belonging to the same origin as itself
- Definition of an origin (URI scheme, Hostname, port)
  - URI Scheme: Protocol (typically http or https)
  - Hostname: domain name (e.g., **example.com**:8080)
  - Port: example.com:**8080** (if unspecified, defaults to 80 for http and 443 for https))

Web Security: Cross-site Scripting

- Cross-site Scripting is executing a foreign (and malicious) piece of code as if it was included in the compromised webpage
- Somehow get the browser to execute a script with the permissions of the attacked domain
  - Non-persistent (disappears ajer page reloads)
  - Persistent (persists across page reloads)
- Most common method: somehow inject JavaScript code into a resource of the attacked domain so that the code executes with the authority of the parent and can access it

Web Security: Cross-site Scripting

- Non-persistent: Occurs when server-side code accepts a query string or form submitted by the user, and sends the string back to the client as a new page or AJAX response without validating it
  - User can inject malicious JavaScript code into the query string or form input (can be hidden)
  - The script when it is sent back now executes with the authority of the server's origin and can access all resources of the same origin at the client

Web Security: Cross-site Scripting

- Persistent: In a persistent XSS attack, the attack string is stored on the server so that future visits to the website (by the same user or different users) would also be subject to the attack
  - Much more devastating than the reflected attacks
  - Result from server not checking the user-specified string before storing it to a database or file (say)

Web Security: Cross-site Scripting

Defense

- Sanitizing user input by checking for JS
  - Hard to do as JS code can be concealed in many ways (e.g., by escaping within HTML or CSS tags)
  - Performance overhead on the server for parsing inputs
- Lighter-weight but incomplete methods
  - Tying cookies to the IP address of the user logged in (works only for XSS attacks that try to steal cookies)
  - Disabling scripts on the page or in a specific section of the page (may prevent legit. scripts from running)
  - New method: Content security policy (allow servers to specify approved origins of content for web browsers) – not yet implemented in all browsers

Web Security: Cross-site Request Forgery

- An attacker attempts to request a URL sent to a user by spoofing it to their benefit
- Relies on the use of reproducible and guessable URLs (typically as parameters of GET requests)
- Cookies are automatically sent with every request, and hence the URL can perform malicious actions on behalf of the client
    - Do not require the server to accept/allow JavaScript code (unlike XSS attacks)

Web Security: Cross-site Request Forgery

Defense

- Make the URL hard to guess by attaching a random nonce or client-specific key to it
  - Works only if nonce/key is not leaked, and is complex
- Things that don’t work, but are often deployed
  - Using POST instead of GET requests (pointless)
  - Using multi-step transactions (makes it harder for the attacker, but they can still forge the sequence)
  - Using a secret cookie (all related cookies will be submitted with every request, even the secret ones)

Exam Logistics

1. HTML and CSS
2. DOM and Events
3. JavaScript
    a. Callback and Closure
    b. ES6: Class, Arrow Functions, Promise
4. AJAX and Node.js
5. Session, Cookie, and Web Security
6. **Exam Logistics**

Exam Logistics

Exam Format

1. Multiple Choice Questions - 20 Questions (30 minutes)
    a. Closed Book
    b. Only your pencil allowed

2. Programming Questions - 5 Questions (2 hours)
    a. Open Book - refer to notes, websites, tutorials etc.
    b. You **must** use your own laptop
    c. Must be done individually
    d. No messaging platforms - i.e., SMS, Instant Messaging, Email
    e. All questions carry equal number of points

Exam Logistics

What to bring

1. Pencil for Optical Sheets
    a. Optical sheet scanner **will NOT recognize answers marked with a pen**

2. Laptop (Fully-charged)
    a. Pre-install **Node.js**, **browser**, and **text editor**
    b. Need to type the code on your laptops with a text editor
    c. Try the code before submitting it
    d. Need to submit the code using a Google Form
    e. We'll only evaluate what's submitted