Document Object Model

CITS3403: Agile Web Development

Unit Coordinator: Tim French

2023, Semester 1
Introduction

• We’ve seen JavaScript core
  – provides a general scripting language
  – but why is it so useful for the web?
• Client-side JavaScript adds collection of objects, methods and properties that allow scripts to interact with HTML documents
  ○ dynamic documents
  ○ client-side programming
• This is done by bindings to the Document Object Model (DOM)
  – “The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.”
  – “The document can be further processed and the results of that processing can be incorporated back into the presented page.”
• DOM specifications describe an abstract model of a document
  – API between HTML document and program
  – Interfaces describe methods and properties
  – Different languages will bind the interfaces to specific implementation
  – Data are represented as properties and operations as methods
• https://www.w3schools.com/js/js_htmldom.asp
The DOM Tree

- DOM API describes a tree structure
  - reflects the hierarchy in the XML document
  - example...

```html
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>A simple document</title>
  </head>
  <body>
    <table>
      <tr>
        <th>Breakfast</th>
        <td>0</td>
        <td>1</td>
      </tr>
      <tr>
        <th>Lunch</th>
        <td>1</td>
        <td>0</td>
      </tr>
    </table>
  </body>
</html>
```
Execution Environment

• The DOM tree also includes nodes for the execution environment in a browser
  • **Window** object represents the window displaying a document
    – All properties are visible to all scripts
    – Global variables are properties of the Window object
  • **Document** object represents the HTML document displayed
    – Accessed through `document` property of Window
    – *Property arrays* for forms, links, images, anchors, ...

• The **Browser Object Model** is sometimes used to refer to bindings to the browser, not specific to the current page (document) being rendered. This includes:
  – Type of browser
  – User’s history
  – Cookies
  – Screen size
  – Location (url)
  – Geolocation
  – Local (browser) storage
DOM Tree in More Detail

Source: tech.irt.org
JavaScript and the DOM

- *Elements* in HTML document correspond to *nodes* on the tree
- These *nodes* bind to JavaScript *Element objects*
- *Attributes* of elements become named *properties* of element node objects
  - `<input type="text" name="address">`
  - The object representing this node will have two properties
    - `type` property will have value “text”
    - `name` property will have value “address”
- Node objects can be addressed in several ways:
  - *arrays* defined in DOM 0
    - forms, elements, images, links,...
    - individual elements are specified by index
      - by *name*
      - by *id*
Method 1: Using DOM Address

- Consider this simple form:
  `<form action = "">
   <input type = "button"  name = "pushMe">
  </form>`

- The `input` element can be referenced (assuming this is the first form in the document) as `document.forms[0].element[0]`

  This example finds the form element with `id="frm1"`, in the forms collection, and displays all element values:

```
Example

var x = document.forms["frm1"];  
var text = ""; 
var i; 
for (i = 0; i < x.length; i++) {
   text += x.elements[i].value + "<br>";
}
document.getElementById("demo").innerHTML = text;
```

Try it Yourself »
Method 2: Using Name Attributes or Type

• Using the name attributes for form and form elements
  – Reference using Java/JavaScript “.” notation

• Example
  
  ```html
  <form name = "myForm" action = "">
    <input type = "button" name = "pushMe">
  </form>
  
  document.myForm.pushMe
  ```

• Referencing the input

• In order to work, all elements from the reference element up to, but not including, the body must have a name attribute

• Names are required on form elements by server-side scripts

• You can also select all elements by tag name.
Method 3: Using ID

- Using `getElementById` with id attributes (cf CSS)
  - id attribute value must be unique for an element

- Example:
  - Set the id attribute of the input element
    ```html
    <form action = "">
        <input type="button" id="on">
    </form>
    ```
  - Then use `getElementById`
    ```javascript
    document.getElementById("on")
    ```
Other Access Methods

• A range of other “short cut” methods may be provided
• Eg. `getElementsByTagName`

```javascript
var tables = document.getElementsByTagName("table");
alert("This document contains " + tables.length + " tables");
```

• Checkboxes and radio buttons have an implicit array, which has their name as the array name

```html
<form id = "topGroup">
    <input type = "checkbox"  name = "toppings"
    value = "olives" />
    ...
    <input type = "checkbox"  name = "toppings"
    value = "tomatoes" />
</form>
```

```javascript
var numChecked = 0;
var dom = document.getElementById("topGroup");
for index = 0; index < dom.toppings.length;index++
    if (dom.toppings[index].checked]
       numChecked++; `
As we’ve seen each element in an HTML document has a corresponding `Element` object in the DOM representation.

The `Element` object has methods to support:
- **Traversing the document**
  - that is, visiting each of the document nodes
- **Modifying the document**
  - for example, removing and inserting child nodes

Various properties of `Element` objects are related nodes, eg:
- `parentNode` references the parent node of the `Element`
- `previousSibling` and `nextSibling` connect the children of a node into a list
- `firstChild` and `lastChild` reference children of an `Element`
  - These would be text nodes or further element nodes contained in the element
- `childNodes` returns a `NodeList` (like an array) of children
function countTags(n) { // n is a Node
    var numtags = 0; // Initialize the tag counter
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // Check if n is an Element
        numtags++; // If so, increment the counter
    var children = n.childNodes; // Now get all children of n
    for(var i=0; i < children.length; i++) { // Loop through the children
        numtags += countTags(children[i]); // Add and recurse on each one
    }
    return numtags; // Return the total number of tags
}
</script>

<!-- Here's an example of how the countTags() function might be used -->

<body onload="alert('This document has ' + countTags(document) + ' tags')"
This is a <i>sample</i> document.
</body>

<!-- From: JavaScript: The Definitive Guide (4th Ed) -->
Example: JavaScript vs DOM

• Blue JavaScript, red DOM...

```javascript
// point anchorTags to a DOM NodeList
var anchorTags = document.getElementsByTagName("a");
// display the href attribute of each element in the NodeList
for (var i = 0; i < anchorTags.length; i++){
    alert("Href of this a element is : " + anchorTags[i].href + "\n");
}
```

DOM Tree Modification

- There are also methods that allow you to modify or construct a DOM tree. eg:
  - The `insertBefore` method inserts a new child of the target node
  - `replaceChild` will replace a child node with a new node
  - `removeChild` removes a child node
  - `appendChild` adds a node as a child node at the end of the children

  you can construct part or whole document dynamically!

- This is what front-end frameworks like Angular or React do: they dynamically build the entire document on the client side.

- Document writing methods include:
  - `open()`
  - `close()`
  - `write()`
  - `writeln()`

```html
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="myApp" ng-controller="customersCtrl">
<ul>
  <li ng-repeat="x in myData">
    {{ x.Name + ', ' + x.Country }}
  </li>
</ul>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('customersCtrl', function($scope, $http) {
  $http.get("customers.php").then(function (response) {
    $scope.myData = response.data.records;
  });
});
</script>
</body>
</html>
```
Example

<script type="text/javascript">
function createNewDoc() {
    var newDoc=document.open("text/html","replace");
    var txt="<html><body>Learning about the DOM is FUN!</body></html>";
    newDoc.write(txt);
    newDoc.close();
}
</script>

<!-- From: http://www.w3schools.com -->
The canvas Element

• The canvas Element
  – Creates a rectangle into which bit-mapped graphics can be drawn using JavaScript
  – Optional attributes: height, width, and id
    • Default value for height and width are 150 and 300 pixels
    • The id attribute is required if something will be drawn

    <canvas id = "myCanvas" height = "200"
            width = "400"> 
        Your browser does not support the canvas element 
    </canvas>

• This can be used to create interactive animations and games in just HTML and javascript:
  
  https://www.w3schools.com/graphics/tryit.asp?filename=trygame_default_gravity
The navigator Object

- Properties of the `navigator` object allow the script to determine characteristics of the browser in which the script is executing.
- The `appName` property gives the name of the browser.
- The `appVersion` property gives the browser version.

```html
<!DOCTYPE html>
<!-- navigate.html
   A document for navigate.js
   -->
<html lang = "en">
  <head>
    <title> navigate.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript" src = "navigate.js" >
    </script>
  </head>
  <body onload = "navProperties()">
  </body>
</html>
```
In addition to the Document Object Model there is also a Browser Object Model (BOM).

This is not supported by a fixed standard, but is a set of features most browsers support, to let developers tailor apps for different browser contexts.

These include:
- Browser type and version (typically misreported)
- The language used in the browser
- The geolocation of the user (https and with user consent)
- The History of the user.
- Any cookies associated with the current domain.

These properties are access through document.navigator.

### Navigator Object Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appName</td>
<td>Returns the name of the browser</td>
</tr>
<tr>
<td>appVersion</td>
<td>Returns the version information of the browser</td>
</tr>
<tr>
<td>cookieEnabled</td>
<td>Determines whether cookies are enabled in the browser</td>
</tr>
<tr>
<td>geolocation</td>
<td>Returns a Geolocation object that can be used to locate the user's position</td>
</tr>
<tr>
<td>language</td>
<td>Returns the language of the browser</td>
</tr>
<tr>
<td>online</td>
<td>Determines whether the browser is online</td>
</tr>
<tr>
<td>platform</td>
<td>Returns for which platform the browser is compiled</td>
</tr>
<tr>
<td>product</td>
<td>Returns the engine name of the browser</td>
</tr>
<tr>
<td>userAgent</td>
<td>Returns the user-agent header sent by the browser to the server</td>
</tr>
</tbody>
</table>

### History Object Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>Returns the number of URLs in the history list</td>
</tr>
</tbody>
</table>

### History Object Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>back()</td>
<td>Loads the previous URL in the history list</td>
</tr>
<tr>
<td>forward()</td>
<td>Loads the next URL in the history list</td>
</tr>
<tr>
<td>go()</td>
<td>Loads a specific URL from the history list</td>
</tr>
</tbody>
</table>
Cookies

- Cookies are a way of websites identifying returning users. As HTTP requests are stateless, the server normally won’t remember any previous requests from a client.
- A cookie is a small text file containing key-value pairs that is stored in the browser.
- The cookie will be sent with a request to the website it is associated with (and only that website).
- Cookies for the current web-page are accessible through the DOM/BOM.
- Cookies are specified with an expiry date or will be deleted when the browser is closed.

```javascript
function setCookie(cname, cvalue, exdays) {
  var d = new Date();
  d.setTime(d.getTime() + (exdays*24*60*60*1000));
  var expires = "expires="+ d.toUTCString();
  document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/;"
}

function getCookie(cname) {
  var name = cname + "=";
  var decodedCookie = decodeURIComponent(document.cookie);
  var ca = decodedCookie.split(';');
  for(var i = 0; i <ca.length; i++) {
    var c = ca[i];
    while (c.charAt(0) == ' ') {
      c = c.substring(1);
    }
    if (c.indexOf(name) == 0) {
      return c.substring(name.length, c.length);
    }
  }
  return "";
}
```
A larger and more secure alternative to cookies is Web Storage (new since HTML5).
This allows a website to store information about a user within the users browser and retrieve it at a later time.
This can be particularly useful for large forms where there is a chance a session could end before the user submits the form.

```javascript
if (localStorage.clickcount) {
    localStorage.clickcount = Number(localStorage.clickcount) + 1;
} else {
    localStorage.clickcount = 1;
}
document.getElementById("result").innerHTML = "You have clicked the button " + localStorage.clickcount + " time(s).";
```
Event-Driven Programming

- **Event-driven programming** or **event-based programming**
  - programming paradigm in which the flow of the program is determined by *sensor outputs* or *user actions* (mouse clicks, key presses) or *messages from other programs*
  - not new - from hardware interrupts to multi-process operating systems to distributed programming to Java listeners to Exceptions...
- **Fundamental to web-based programming**
  - client-server model
  - stateless programming
  - controlled from browser (user) end
- Event driven programming drives many of the technologies we will cover in this unit:
  - Sockets
  - AJAX
  - Javascript callbacks
Event-Driven Programming

- **Batch program**

  read a number (from the keyboard) and store it in variable A[0]
  read a number (from the keyboard) and store it in variable A[1]
  - *synchronous* (program waits for input)

- **Event-driven program**

  set counter K to 0
  repeat {
    if a number has been entered (from the keyboard) {
      store in A[K] and increment K
      if K equals 2 print A[0]+A[1] and reset K to 0
    }
  }
  - *asynchronous* (program polls for input)
Event-Driven Programming

- Program “loop” divided into two distinct tasks
  - event detection
  - event handling
- Application programmer may be freed from event detection (and hence loop) in a number of ways
  - embedded programs may use interrupts - handled by hardware (no loop needed)
  - programming environment or execution environment may do this for you - in our case the browser
    - allows programmer to focus on event handling
- Browser “listens” (polls or interrupts) for events
  - user actions (eg. <enter>, mouse clicks, ...)
  - server responses (eg. page loaded, AJAX responses, calculation, ...)
- When it recognises an event, it invokes the appropriate code to handle the event (event handler), passing information about the event as required
- But how does the browser know what code to call?
- For the browser to know what code to invoke for different actions, code elements must be registered with, or bound to, events
- What defines the events, their meanings, and parameters?
  - the DOM!
Event Registration

• DOM 0 provides two ways to register an event handler:

1. Assign the event handler script to an event tag attribute

   <input type = "button" id = "myButton"
   onclick = "alert(‘You clicked my button!’);" />

   onclick is a tag attribute for the button “click” event

   Usually the handler script is more than a single statement and called as a function:

   <input type = "button" id = "myButton"
   onclick = "myButtonHandler();" />

2. Assign the event handler to the appropriate property of the element’s object

   <input type = "button" id = "myButton" />
   .
   .
   .

   document.getElementById("myButton").onclick = myButtonHandler;

   — statement must follow both handler function and form element so (JavaScript) interpreter has seen both
   — note: just function name, not function call (or string)
### Events and their Tag Attributes

<table>
<thead>
<tr>
<th>Event</th>
<th>Tag Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>blur</td>
<td>onblur</td>
</tr>
<tr>
<td>change</td>
<td>onchange</td>
</tr>
<tr>
<td>click</td>
<td>onclick</td>
</tr>
<tr>
<td>dblclick</td>
<td>ondblclick</td>
</tr>
<tr>
<td>focus</td>
<td>onfocus</td>
</tr>
<tr>
<td>keydown</td>
<td>onkeydown</td>
</tr>
<tr>
<td>keypress</td>
<td>onkeypress</td>
</tr>
<tr>
<td>keyup</td>
<td>onkeyup</td>
</tr>
<tr>
<td>load</td>
<td>onload</td>
</tr>
<tr>
<td>mousedown</td>
<td>onmousedown</td>
</tr>
<tr>
<td>mousemove</td>
<td>onmousemove</td>
</tr>
<tr>
<td>mouseout</td>
<td>onmouseout</td>
</tr>
<tr>
<td>mouseover</td>
<td>onmouseover</td>
</tr>
<tr>
<td>mouseup</td>
<td>onmouseup</td>
</tr>
<tr>
<td>reset</td>
<td>onreset</td>
</tr>
<tr>
<td>select</td>
<td>onselect</td>
</tr>
<tr>
<td>submit</td>
<td>onsubmit</td>
</tr>
<tr>
<td>unload</td>
<td>onunload</td>
</tr>
</tbody>
</table>

```html
<!DOCTYPE html>
<html>
<body>
<p>Click the button to display the date.</p>
<button onclick="displayDate()">The time is?</button>
<script>
function displayDate() {
    document.getElementById("demo").innerHTML = Date();
}
</script>
<p id="demo"></p>
</body>
</html>
```

Click the button to display the date.

Wed Mar 13 2019 13:12:10 GMT+0800
Tag Attributes and their Tags

- Most event tag attributes can appear in several tags
- Meaning (action) depends on both the tag attribute \textit{and} the tag in which it appears. Eg.
  - an element gains “focus” when the mouse is passed over it and left clicked, or user tabs to element
  - lose focus when it passes to another element - called \textit{blurring}
    - different meaning (action) for \textlt{a} and \textlt{textarea}
<body onload="load_greeting();">
  <p /></body>

function load_greeting () {
  alert("You are visiting the home page of
  Pete's Pickled Peppers \n  Welcome!!!");
}
Mouseover events

- Any HTML element can be have a mouseover event associated with it.

```html
<!DOCTYPE html>
<html>
<body>
<div onclick="mOver(this)" onmouseout="mOut(this)"
style="background-color:#D94A38;width:120px;height:20px;padding:40px;">
Mouse Over Me</div>
<script>
function mOver(obj) {
  obj.innerHTML = "Thank You"
}

function mOut(obj) {
  obj.innerHTML = "Mouse Over Me"
}
</script>
</body>
</html>
```
Handling Events from Text Box and Password Elements

- An important use of events is to validate the content of forms, without using bandwidth and time to access a remote server.
- By manipulating the focus event the user can be prevented from changing the amount in a text input field.

Note: this is possible to work around.

- Copy the page but leave out the validation code.
- Simulate an HTTP request directly with socket-level programming.

If the validity of data is important, the server needs to check it.

JavaScript Example

```javascript
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == ") {
        alert("Name must be filled out");
        return false;
    }
}
```

The function can be called when the form is submitted:

HTML Form Example

```html
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>
```
DOM 2 Event Model

- DOM 2 is defined in *modules*
- The *Events* module defines several submodules
  - *HTMLEvents* and *MouseEvents* are common
- An event object is passed as a parameter to an event handler
  - Properties of this object provide information about the event
  - Some event types will extend the interface to include information relevant to the subtype. For example, a mouse event will include the location of the mouse at the time of the event

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>blur</td>
<td>The event occurs when an element loses focus</td>
<td>FocusEvent</td>
</tr>
<tr>
<td>canplay</td>
<td>The event occurs when the browser can start playing the media (when it has buffered enough to begin)</td>
<td>Event</td>
</tr>
<tr>
<td>canplaythrough</td>
<td>The event occurs when the browser can play through the media without stopping for buffering</td>
<td>Event</td>
</tr>
<tr>
<td>change</td>
<td>The event occurs when the content of a form element, the selection, or the checked state have changed (for <code>&lt;input&gt;</code>, <code>&lt;select&gt;</code>, and <code>&lt;textarea&gt;</code>)</td>
<td>Event</td>
</tr>
<tr>
<td>click</td>
<td>The event occurs when the user clicks on an element</td>
<td>MouseEvent</td>
</tr>
<tr>
<td>contextmenu</td>
<td>The event occurs when the user right-clicks on an element to open a context menu</td>
<td>MouseEvent</td>
</tr>
<tr>
<td>copy</td>
<td>The event occurs when the user copies the content of an element</td>
<td>ClipboardEvent</td>
</tr>
<tr>
<td>cut</td>
<td>The event occurs when the user cuts the content of an element</td>
<td>ClipboardEvent</td>
</tr>
<tr>
<td>dbldclick</td>
<td>The event occurs when the user double-clicks on an element</td>
<td>MouseEvent</td>
</tr>
<tr>
<td>drag</td>
<td>The event occurs when an element is being dragged</td>
<td>DragEvent</td>
</tr>
<tr>
<td>dragend</td>
<td>The event occurs when the user has finished dragging an element</td>
<td>DragEvent</td>
</tr>
<tr>
<td>dragenter</td>
<td>The event occurs when the dragged element enters the drop target</td>
<td>DragEvent</td>
</tr>
<tr>
<td>dragleave</td>
<td>The event occurs when the dragged element leaves the drop target</td>
<td>DragEvent</td>
</tr>
</tbody>
</table>
Event Flow

- DOM 2 defines a process for determining which handlers to execute for a particular event.
- The event object representing the event is created at a particular node called the **target node**.
- The process has three phases...
  - In the **capturing phase** each node from the document root to the target node, in order, is examined.
    - If the node is not the target node and there is a handler for that event at the node and the handler is enabled for capture for the node, the handler is executed.
  - Then all handlers registered for the target node, if any, are executed.
  - In the **bubbling phase** each node from the parent of the target node to the root node, in order, is examined.
    - If there is a handler for that event at the node and the handler is **not** enabled for capture for the node, the handler is executed.

Some event types are not allowed to bubble: load, unload, blur and focus among the HTML event types.
Event Propagation

- As each handler is executed, properties of the event provide context
  - The `currentTarget` property is the node to which the handler is registered
  - The `target` property is the node to which the event was originally directed
  - `currentTarget` is always the object listening for the event; `target` is the actual target that received the event
- One major advantage of this scheme over DOM 0 is that event handling can be centralized in an ancestor node
- For example, a calculator keyboard will have a number of digit buttons
  - In some GUI frameworks, a handler must be added to each button separately
  - In DOM 2, the buttons could be organized under a single node and the handler placed on the node

```javascript
document.getElementById("myP").addEventListener("click", myFunction, true);
document.getElementById("myDiv").addEventListener("click", myFunction, true);
```
Event Handler Registration

- Handlers are called **listeners** in DOM 2
- `addEventListener` is used to register a handler, it takes three parameters
  - A string naming the event type
  - The handler
  - A boolean specifying whether the handler is enabled for the capture phase or not

```html
<p>A function is triggered when the user is pressing a key in the input field.</p>
<input type="text" onkeydown= f(event)"
<script>
function f(e) {
    alert("You hit the "+e.keyCode+" key");
}
</script>
```

```javascript
window.addEventListener("keydown", moveSomething, false);

function moveSomething(e) {
    switch(e.keyCode) {
        case 37:
            // left key pressed
            break;
        case 38:
            // up key pressed
            break;
        case 39:
            // right key pressed
            break;
        case 40:
            // down key pressed
            break;
    }
}
```