305CDE Worksheet 4

About

This week's lab tasks will focus on two important areas where JS is used on the client side:

- interacting with the DOM,
- and asynchronously getting or setting data on remote servers, often referred to as AJAX.

Note that AJAX technically uses XML as the data format, but we will be using JSON which is easier to parse, and becoming the defacto standard for much data on the web. The principles of asynchronous communication and client side *callback functions* are effectively identical for XML and JSON though.

Resources

- Introduction to the DOM
- The DOM API
- Using native JSON
- Getting Started with AJAX

Task List

The tasks this week are split in to two "tracks" depending on your confidence with JS.

- Follow "Track 1" if you are still building confidence with functions and objects.
- Follow "Track 2" if you were able to complete most of the tasks on the previous weeks' worksheets (and wish to learn more advanced techniques for your Challenge assigments).

Since this material is more advanced than previous weeks, aim to get as much understanding as possible during 60-80 minutes of lab time. Follow this up in your own time with trying to complete at least the tasks in "Track 1"

NOTE: Remember to commit your changes to a fork of the 305CDE git repository (see last week's worksheet) so that you can continue later. The techniques and knowledge developed this week will be very useful later.

Track 1

DOM:

1. `events.html`
2. `map.html`

AJAX:

3. `JSON.html`
4. `ajax_basic.html`

Track 2

The JavaScript on this track is a little more complex.

DOM:

1. `dom_scripting.html` and `js/dom_utils.js` or `js/dom_utils_global.js`

AJAX:

2. `ajax_sync.html` and `ajax_utils.js`

Track 1

1. DOM events

You have already seen events in JS: `window.onload` for example, is fired when the window has completed loading. Most interactive JS requires event handlers for a number of types of events. A handler is just a piece of code (typically a function) that is run when the event happens.

Most events are context-specific: that means they relate to the DOM element that the event involved, for example a `<button>` or a `<div>` with a particular `id` or `class` attribute.

You also need to know how to "turn off" event handlers that you no longer need.

The code you will look at below invokes handlers that check for methods and properties that are not shared across all browsers (browser incompatibility implies that *the same JS does not work in all browsers*). Libraries such as jQuery do this automatically, but it is important that you see how it is achieved: loading all of jQuery just to handle a click event is usually massive overkill, it is often better to keep the dependencies to a minimum.

- Load `events.html` in Brackets and preview its functionality.
- Note that the `window.onload` event is given a function (this should be familiar from previous weeks)
  - Observe that the function checks whether particular methods are available on `div` in the DOM
  - Depending on the method availability, the `onload` handler invokes a click handler on the `div` and *passes it a callback function* called `handleClick`
- Note that the callback `handleClick` looks for a property called `target` on the event it is passed
  - Depending on the property availability, the callback correctly identifies the target element in the DOM

Test your understanding

- Add a new `div` containing some text
- Add a "mouseover" event listener on your new `div`
- Add a mouseover handler which alerts the user which element ID was hovered over

2. DOM elements

The DOM is essentially a description of the elements on a web page along with their attributes, properties, and accessible methods. See the references above for a more detailed run through of what the DOM is. To see JS interacting with the DOM, follow the next steps.

- Load `map.html` and preview the functionality via Brackets
- Observe that the `showCoords()` function sets the `innerHTML` property of a DOM element
- Also observe that the `addtext()` function *dynamically adds a new element to the DOM* (a "text node").
  - Similarly you can add other HTML tags such as `<a>` or `<p>` etc., or update their attributes

Test your understanding

Do these after a quick scan of the MDN DOM introduction.

- Use the `createElement` method of the `document` object to add a new list `<ul>` to the DOM

- Populate this list using a JS `for` loop which uses `appendChild` to add list items `<li>`, using the new `<ul>` as the parent node.
- Use a nested `innerHTML` assignment to put some text in your list items.

3. JSON

See the MDN Using native JSON guide to help understanding the two key JSON methods described here.

JSON is the "JavaScript Object Notation" and is a way of describing structured data which is particularly useful in JS because it corresponds very closely to the object literal syntax.

It's two main methods are for converting strings to and from "native" JSON objects that your JavaScript program can work with.

- Load `JSON.html` and preview it, making sure to view the console output using F12.
- Note that the first output corresponds to accessing a property of a JS object that was *created* from a string using `JSON.parse()`.
- Note that the second output corresponds to displaying in text format (also suitable for transmission over HTTP) the contents of a JS object, including its property names and values.

Test your understanding

- Define a new JS object in the `JSON.html` script which also contains *nested objects* (recall that JS objects can be nested multiple times) as some of its properties. For example:

```
var employee = {
    name: { first: "Colin",
            second: "Stephen" },
    title: "Assistant Lecturer"
}
```

- Convert your object to a JSON string `myString` and display it in a `<div>` somewhere on the page
- Convert the string *back to a JS object* `myObject` and assign one of the *nested* objects to a new variable. For example:

```
var employeeName = myObject.name;
```

- Stringify the new subobject and display it in another `<div>` elsewhere on the page.

4. AJAX in different browsers

AJAX essentially uses the JavaScript `XMLHttpRequest` object to communicate with server-side scripts. It can send as well as receive information in a variety of formats, including JSON, XML, HTML, and even text files.

In this section we will see the main "workflow" involved in making an AJAX request and processing a response using a *callback function*. The code also takes account of browser inconsistencies, which is usually done automatically if you are using jQuery, for example.

- Load the `ajax_basic.html` file in Brackets and preview it with live preview
  - **NOTE** If you do not use live preview the AJAX call will fail due to cross domain security restrictions used by most servers/browsers
- Note that clicking on the "ajaxButton" will call a function that makes a request to a certain URL
- The `makeRequest()` call first checks which suitable HTTP request methods are available in the browser
  - In *modern* browsers, `window.XMLHttpRequest` will be available. This is the standard approach to AJAX if you don't need to support older browsers.
- Having set an appropriate `httpRequest` object, the code assigns a callback called `alertContents` to an event listener `onreadystatechange` on the object that is triggered when the actual HTTP request completes and returns some data or an error.
- Finally the `httpRequest` objects calls its `open()` and `send()` methods to get the data from the given URL.

The `alertContents` function also uses the `httpRequest` object (note that it is defined within an immediately invoked function expression!):

- The alert checks whether the new `readyState` on the request object corresponds to the completion of the request (value `=== 4`)
- If so, and if the server response status indicates success (value `=== 200`), then the `responseText` from the HTTP response is shown in an alert.

Test your understanding

You can make a simple "404" checker as follows.

- Add a text input element to the page which the user can type a URL in to.
- Add a button that when clicked tries to do an AJAX http request to the given URL.

- Define a new callback function, similar to `alertContents` which checks whether the requested URL has a “404” status or not.
- If it does, pop up an alert saying “Site is DOWN!”
- Adjust the appropriate event listener in the `makeRequest` function of `ajax_basic.html`, so that your new callback is set as the handler for all HTTP requests made by your checker page.

Track 2

1. DOM Scripting

There is a lot going on “behind the scenes” in libraries such as jQuery which abstract away many of the browser inconsistencies and deep hierarchies of objects, properties and methods relating to the DOM. You can review these DOM objects and their functionality on the the MDN DOM API documentation.

This task involves a “helper script” that does some DOM and event manipulation for you. It is like a “mini jQuery” and even defines a utility DOM selector object called `$`.

- Load up `dom_scripting.html` and `js/dom_utils.js` in Brackets and check the functionality.
- Three things are being done:
    1. A button is assigned an event handler (callback) which fires every time the button is clicked
    2. A button is assigned a self-removing event handler which fires the *first* time the button is clicked
    3. A button is assigned a DOM manipulating event handler for its click events

The actual code in `dom_scripting.html` is short and self contained, this is because the selectors `U.$()`, the event listeners and unlisteners `U.addEvent()` and `U.removeEvent()`, and the DOM manipulator `U.setText()` are defined in the separate utility code file `dom_utils.js`.

- Have a look at the utility JavaScript
- Note that `U` is simply an object with some useful methods which wrap native JS methods for achieving event interaction and DOM manipulation. An important thing to note is that where there exist browser incompatibilities, the utility methods defined on `U` need to conditionalise on the JS methods being available. For example, in `U.setText()`:

```
if (output.textContent !== undefined) {
    // use this property
} else {
    // use innerText instead
}
```

Test your understanding

- Have a look at `dom_utils_global.js`, which is an implementation of a few additional DOM/event helpers
  - In particular note that it is defined as a module.
  - Carefully think about how `$live` is defined here:
    - an immediately invoked function returns a (closure) function which adds handlers to events and which stores these associations in an object containing event handler arrays
    - in addition, the `dispatchEvent()` is a closure over the `eventRegistry` object and is therefore able to be called against this registry whenever a listened-to event happens, *even when the event happens on an element that was not defined in the DOM when the event handler was added!*
    - this is the power of closures
- Adjust `dom_scripting.html` to include the `dom_utils_global.js` file (as well as the original `dom_utils.js`)
- Now refactor the JS code in the HTML file to use `window.$live` to register the events and handlers previously dealt with by the `U` object.
- Try adding a DOM element dynamically to the page, which has a "live" listener already defined earlier in your code. It should react to the listened-to event even though it was not present when you added the `$live` event association.

2. AJAX processing

NOTE: Also take a look at "Track 1" Part 4 to see how to conditionalise on the `XMLHttpRequest()` method being present.

In this example you will do some first steps in JSON fetching, parsing, and acting on the information retrieved.

- Load up `ajax_sync.html` and `js\ajax_utils.js` and preview the output.
- Also look at the Chrome developer tools 'Network' tab and refresh the page, to see the various JSON files that come from the server
- Note that the HTML file contains a `try` block which does something very simple:

  - gets the result of `getJsonSync` on a particular URL
  - adds an attribute `heading` of the story data to the page
  - then cycles through a list defined in another data attribute to perform the same operations as above: get the result of a URL, and add one of its attributes to the page.

Again, most of the work is in the utility file!

- Look at the `ajax_utils.js` file.
- Note there are two main blocks of functionality (ignoring the fake network delay)
  1. AJAX stuff
  2. DOM stuff
- The `getSync()` function basically defines a new request object, opens the provided URL using this object, and sends the 'GET' request out
- After some possible waiting time, the request will complete so the status is checked (200 means "OK") and the data in the request response `req.response` is returned to the caller

Since we assume that `getSync` actually returns a JSON string in this case, there is an additional `get` function called `getJsonSync`. This simply "wraps" the regular `getSync` call with a function that parses the JSON data and returns a JavaScript object instead, ready for use in our calling code back in the HTML file.

- Review the DOM stuff code at the bottom of the file
- Note that this is straightforward: wrap some element creation and content setting in some function blocks that we can call quickly from elsewhere.

Test your understanding

- Parse the content of the `chapter` objects returned by the `getJsonSync` as they come in from the server:
  - If the html attribute contains the word "dictum" then make a AJAX call to a dictionary API that returns JSON, such as one of the API endpoints listed on the Wordnik API documentation
  - Parse the JSON response for a definition of the word and dynamically add a DOM element to the HTML page containing the term "dictum" along with the returned definition.
- Now make your AJAX call and JSON response parsing dynamic, by allowing the user to double-click any word on the screen: double clicking

it should provide a new entry in the “word dictionary list” somewhere on the page, containing the definition or “not found”. You may find this StackOverflow question on handling double-clicks on words useful.