305CDE Lab 4

The DOM and AJAX

October 2014

Overview

- Functions III (Callbacks)
- Document Object Model (DOM)
- JavaScript Object Notation (JSON)
- Asyncronous JavaScript And ~~XML~~ JSON (AJAX)

Functions III

Callbacks

Remember that functions are *first class objects* in JS. So they can be used as:

- Return values from other functions
- Parameter values ("callbacks") for input to other functions

This is very handy in an asynchronous event-based language such as JS. You can use callbacks to "react" to an event when it occurs:

- Associate *event listeners* with *handler functions* using callbacks.

Passing Functions as Arguments

Look at the following use of a callback:

```
function some_function(arg1, arg2, callback) {
  var my_number = Math.ceil(Math.random() *
    (arg1 - arg2) + arg2);
  // more code here - may take a while
  callback(my_number);
}
some_function(5, 15, function(num) {
  console.log("callback called! " + num);
});
```

- Third parameter to `some_function()` is a function
- This "callback" is *invoked* inside `some_function`
- So *any functionality* can be achieved at this point:
  - just pass in a different callback function

Asynchronous Callbacks

This is most useful when you wish to execute code asynchronously. Suppose `some_function()` takes a long time to complete its operation. Using a callback parameter implies:

- No need to wait for `some_function()` to `return` a value.
- Instead, use the callback to describe what to do when `some_function()` completes.
  - Usually *invoke the callback* somehow, towards the end of the function block
- Can then *immediately* get on with the next bit of the program, without waiting.

This is why callbacks are used extensively in JS, especially during AJAX calls which can involve slow network responses.

Document Object Model (DOM)

What is the DOM

1. Structured tree representation of HTML and XML "documents":
   - Nodes
   - Objects
2. Programming interface (API) for interacting with these:
   - Properties
   - Methods

Used by programs to dynamically change documents:

- Structure
- Style
- Content

The DOM is the connection between web pages (HTML/XML) and scripts or programming languages.

DOM Objects

All of the properties, methods, and events available for manipulating and creating web pages are organized into DOM objects. For example:

- the `document` object that represents the document itself
- the `table` object that implements the special `HTMLTableElement` DOM interface for accessing HTML tables
- and so forth

HTML page content is stored in DOM objects and may be accessed and manipulated via JavaScript, or by other languages such as Python.

Accessing the DOM

You have already seen ways to access the `document` or `window` elements of the DOM:

```
body.onload = function() {
  window.alert("Body of page loaded!");
};
```

The same applies for any element defined in the DOM:

- access the DOM object property or method just like with any other JS object property or method
- usually use the "dot" notation as we do here

Add Elements to the DOM

You can dynamically add nodes to the DOM tree describing an HTML document:

```
window.onload = function() {
  // create a couple of elements
  heading = document.createElement("h1");
  heading_text = document.createTextNode("My Title!");
  heading.appendChild(heading_text);
  document.body.appendChild(heading);
}
```

Key DOM Objects

`Document` and `window` objects are the objects whose interfaces you generally use most often in DOM programming.

- `window`: represents something like the browser
- `document`: the root of the document itself
- `Element`: inherits from the generic `Node` interface, and together these provide many of the methods and properties you use on individual elements.

DOM `window` and `document`

Common API uses include:

- `document.getElementById(id)`
- `document.createElement(name)`
- `window.content`
- `window.onload`
- `window.scrollTo`

DOM `element` and `node`

Common API uses include:

- `element.getElementsByTagName(name)`
- `parentNode.appendChild(node)`
- `element.innerHTML`
- `element.style`
- `element.setAttribute`
- `element.getAttribute`
- `element.addEventListener`

JSON

Examples

JS Object:

```
var employee = {
  name: {
    first: "colin",
    second: "stephen"
  },
  title: "Assistant Lecturer"
}
```

JSON String:

```
var employeeString = '{
  "name": {
    "first": "colin",
    "second": "stephen"
  },
  "title": "Assistant Lecturer"
}'
```

Serialising Data in JS

JSON is a syntax for serializing

- objects
- arrays
- numbers
- strings
- booleans
- null

It is based on JavaScript syntax but is distinct from it: some JavaScript is not JSON, and some JSON is not JavaScript.

NOTE: JSON doesn't natively represent more complex data types like functions, regular expressions, dates, and so on.

JSON Methods

`JSON.parse()`

- Parse a string as JSON.
- Optionally transform the produced value and its properties.
- Return the value as a JS object.

`JSON.stringify()`

- Return a JSON string corresponding to the specified JS value
- Optionally include only certain properties, or replace property values in a user-defined way.

Asyncronous JavaScript And ~~XML~~ JSON (AJAX)

What is AJAX

- Basically it is the use of the JS `XMLHttpRequest` object to communicate with server-side scripts.
- It can send as well as receive information in a variety of formats:
  - JSON, XML, HTML, and even text files.

AJAX's most appealing characteristic is its "asynchronous" nature:

- Update local portions of a page based upon user events
- Make requests to the server without reloading the page
- Receive and work with data from the server

Making an HTTP Request

To make an HTTP request, you need a JS object with a suitable method:

```
var httpRequest;
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
    httpRequest = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE 8 and older
    httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
```

After this the `httpRequest` object can be used to make AJAX calls.

Process Responses to Requests

Before making the actual HTTP request with the object:

- Decide what to do with the response that will be coming back.

```
httpRequest.onreadystatechange = myResponseHandler;
// OR
httpRequest.onreadystatechange = function(){
    // process the server response
};
```

Send the HTTP Request

Once you have set what will be done with the response, you need to make the request.

```
httpRequest.open('GET', url, true);
httpRequest.send(null);
```

`open()` takes:

- the method 'GET', 'POST', etc.
- the URL to request
- whether the request is asynchronous or not

`send()` takes:

- optional data for a 'POST' request - a JSON string is a good choice!

Handling the Response

You need a response handler to handle `onreadystatechange` events fired by changes coming from the server. The handler should check:

```
if (httpRequest.readyState === 4) {
    // everything is good, the response is received
} else {
    // still not ready
}
```

- The `readyState` can be:
  - 0 (uninitialized)
  - 1 (loading)
  - 2 (loaded)
  - 3 (interactive)
  - 4 (complete)

Checking Response Code

There are lots of possible HTTP response codes indicating success or otherwise of a request. You need to check:

```
if (httpRequest.status === 200) {
    // perfect!
} else {
    // oh dear :-(
}
```

Once checked, you can use:

- `httpRequest.responseText`: returns the server response as a string of text
- `httpRequest.responseXML`: returns the response as an `XMLDocument` object you can traverse using the JavaScript DOM functions

Example onreadystatechange Handler

```
function myResponseHandler() {
  if (httpRequest.readyState === 4) {
    if (httpRequest.status === 200) {
      var response = JSON.parse(httpRequest.responseText);
      alert(response.computedString);
    } else {
      alert('There was a problem with the request.');
    }
  }
}
```