6. Client-Side Processing

6.1. Client-side processing

- client program (e.g. web browser) can be used to
  - customise interaction with the user
  - validate user input (although HTML5 now provides this)
  - generate (part of) document dynamically
  - send requests to a server in the background
  - make interaction with a web page similar to that of a desktop application
- this is typically done using *JavaScript*, since a Javascript interpreter is built into browsers

6.2. JavaScript

- interpreted, scripting language for the web
- loosely typed
    - variables do not have to be declared
    - the same variable can store values of different types at different times
- HTML `<script>` element specifies script to be executed
    - `type` attribute has value `text/javascript`
    - `src` attribute specifies URI of external script
    - usually `<script>` appears in the `<head>` and just declares functions to be called later
- ECMAScript, 10th edition (June 2019) is the latest standard

6.3. Client-side scripting

- Javascript scripts can be executed upon, e.g.,
    - a `script` element being encountered in a document (not recommended)
    - event occurrences
- many different events (18 in HTML 4, many more in HTML 5):
    - window events: `load`, `unload`
    - mouse events: `click`, `mouseup`, `mousemove`
    - keyboard events: `keydown`, `keyup`
    - form events: `submit`
    - media events: `play`

6.4. Document Object Model (DOM)

- DOM is a W3C recommendation (levels 1, 2, 3 and 4)
- defines *API* for HTML and XML documents
- defines *logical* structure (model) of documents
- document modelled as a *tree* (or *forest*) of nodes
- using DOM, programmers can
    - build documents
    - navigate their structure
    - add, modify, delete elements and content
- purpose is to provide *portability* across web browsers
- DOM is *platform-neutral* and *language-neutral*
- language bindings for *JavaScript*, among others

6.5. Document methods and properties

- The following

Document title: Client-Side Processing*Document title*: Client-Side Processing

    was generated by placing the following script

    ```
    <script type="text/javascript">
      document.write("<em>Document title</em>: ", document.title);
    </script>
    ```

    inside an HTML table cell (not recommended)
- `document` is a special *object* referring to the document displayed in the browser window
- `write` is a *method* defined on an HTML `document`, which writes text into it

- `title` is a *property* defined on `document`, which retrieves the title

6.6. Events

- more commonly, scripts are executed in response to events, e.g, clicking a button
- the following button Click to see a message was produced using (old style):

```
<button onClick="window.alert('Hello World!')">
  Click to see a message
</button>
```

- the `button` element creates a button
- what is displayed on the button is specified by the contents of the `button` element
- `onClick` is an *event attribute*
- `alert` is a *method* of `window`, which opens a new window displaying the message given as argument
- here the script is embedded in the value of the `onClick` attribute
- best practice dictates that HTML and Javascript should be separated

6.7. Calling a function

- say we want to call a Javascript function when a button is clicked
- we can use the following code (still old style):

```
<button onclick="tableOfSquares()">Click to produce table of squares</button>
```

- this will produce the button Click to produce table of squares which calls the user-defined function `tableOfSquares`
- we need somewhere on the page to write the table
- we use an empty `li` element with `id="tos"`
-

6.8. Table of squares function

- the following function produces a table of squares of the first *n* integers, where *n* is entered by the user:

```
function tableOfSquares() {
  // Display a prompt with a zero in it
  var num = window.prompt("Enter an integer", "0");
  var myTable = "<table class='border figure'>";
  var count = 0;
  while(count < num) {
    // Each row of the table is an integer and its square
    myTable = myTable + "<tr><td>" + count + "</td><td>"
                      + count*count + "</td></tr>";
    count++;
  }
  document.getElementById("tos").innerHTML = myTable + "</table>";
}
```

6.9. Comments on previous script

- `var` declares a *variable*
- variables do not need to be declared before use, nor are they statically typed

- a variable can be initialised with a value, e.g., `count` is set to zero
- a JavaScript comment is indicated by `//`
- `prompt` is a *method* of `window`, which opens a dialog box with the given message and default value for the input; it returns the value entered by the user
- `myTable` is used to store a string representing an HTML table
- JavaScript provides the usual `while` and `for` loop constructs
- `+` is used for string concatenation
- `count++` adds one to the value of the variable `count`
- `getElementById` is a document *method* returning a reference to the identified element
- `innerHTML` is a property of DOM nodes which can be set by assigning to it a string representing HTML

6.10. Defining a function

- function `tableOfSquares` has to be defined somewhere, either
  - in a `script` element in the `head` of the document, or
  - in an external file with the extension `js`
- in our case, all functions are defined in the external file `client.js`
- this file is referenced in the `head` of this document as follows:

```
<script type="text/javascript" src="client.js" />
```

6.11. Event handlers

- the value of an `onclick` attribute is an *event handler*
- rather than embedding the event handler code in the HTML, it is preferable to
  - *bind* the `click` event to the button, and
  - associate a handler function with the event

  in Javascript code separate from the HTML
- now the code for the button (which we need to be able to identify) is:

```
<button id="hello-button">Click to produce message</button>
```

- the code for binding the `click` event to the button and associating an event handler function is:

```
document.getElementById("hello-button").onclick = helloAgain;
```

- note that `helloAgain` is simply the *name* of the function to be called

6.12. Event handlers example

- one problem is that the element with id `hello-button` is only available *after* the page has loaded
- so we need to delay binding the `click` event to the button, as follows:

```
window.onload = function() {
                  document.getElementById("hello-button").onclick = helloAgain;
                }
```

- this binds the `load` event to the `window` object
- it also associates an *anonymous* function with the event
- when button Click to produce message is clicked, the event handler calls the user-defined function `helloAgain`
- function `helloAgain` is defined as follows:

```
function helloAgain() {
  window.alert("Hello again world!");
}
```

6.13. Event listeners

- rather than setting the `onclick` property (attribute) of an element:

  ```
  document.getElementById("hello-button").onclick = helloAgain;
  ```

- we could add an *event listener* to the button element as follows:

  ```
  document.getElementById('hello-button').addEventListener('click', helloAgain);
  ```

- this calls `helloAgain` when the `click` event occurs
- this provides the most flexible form of event handling, e.g.:
  - multiple functions can be called for a single event
  - event listeners can be removed using `removeEventListener`

6.14. Functions and form fields

- Enter a word:

- the above was generated by the following:

  ```
  <form>
    <label>Enter a word:</label>
    <input type="text" id="myWord" />
    <input type="button" id="myButton" value="Translate" />
    <input type="text" id="myResult" />
  </form>
  ```

- the `form` element indicates an HTML form
- an HTML form is usually used for submitting information to a server, but not here
- the `input` element (with `type="text"`) creates a single-line textbox
- the `input` element (with `type="button"`) creates a button with the given `value` displayed on it
- `myWord`, `myButton` and `myResult` identify the `input` elements of the `form`
- in the Javascript file

  ```
    document.getElementById("myButton").onclick = myTranslate;
  ```

  will call `myTranslate` (next slide) when the button is clicked

6.15. Defining the function myTranslate

- function `myTranslate` is defined (in `client.js`) as follows:

  ```
  function myTranslate() {
     var word = document.getElementById("myWord").value;
     var result = "unknown";
     if (word === "hello")
        result = "buongiorno";
     else if (word === "goodbye")
        result = "arrivederci";
     document.getElementById("myResult").value = result;
  }
  ```

- `word` contains the word the user entered

- `value` refers to the contents of the identified `input` elements
- JavaScript has two equality operators: == and ===
- == tests if two values are equal (possibly after type coercion)
- === tests if both the types and values are equal, so is considered safer
- e.g., `(1 == true)` is true, while `(1 === true)` is false

6.16. Navigating the DOM tree

- each DOM `node` object has a number of properties for navigation, e.g.:
  - `firstChild`
  - `nextSibling`
  - `parentNode`
  - `childNodes`

  which return, respectively, the *first child*, *next sibling*, *parent* and *all children* of the node
- other properties include
  - `nodeName`
  - `nodeValue`

  for returning the name of an element node and the textual content of a text node, respectively
- often easier to navigate using the `getElementsByTagName` method, which takes an element name as argument and returns a collection of all element nodes with that name

6.17. Finding elements by name

- say we want to output the value of all `h1` elements from the current document
- we want them to appear as a list below when the following button is clicked: Click for headings

6.18. Finding elements by name (function body)

- we can use the following script:

```
var headings = document.getElementsByTagName("h1");
var output = "<ul>";
for ( i = 0; i < headings.length; i++ )
    output = output + "<li>" + headings[i].firstChild.nodeValue + "</li>";
document.getElementById("headingList").innerHTML = output + "</ul>";
```

- `getElementsByTagName` returns a *collection* of nodes
- `length` is a *property* of a *collection*; it returns the number of items in the collection
- the *i*'th item in a *collection* can be retrieved using `item(i)` as a method call or using array-like indexing
- `firstChild` is a *property* of a node; it returns the *first child* of the node if it exists, otherwise it returns null
- `nodeValue` is a *property* of a node; it returns the *text value* of the node if it is a *text* node, otherwise it returns null
- an element with textual contents is represented by an element node having a single text node as a child

6.19. Using CSS selectors

- the DOM also provides the methods
  - `querySelector`
  - `querySelectorAll`

which take CSS selectors as argument and are more flexible

- `querySelector` returns the first matching node, while `querySelectorAll` returns all matching nodes
- instead of `getElementsByTagName("h1")` we could use `querySelectorAll("h1")`
- instead of `getElementById("headingList")` we could use `querySelector("#headingList")`
- but we could also use selectors
    - `div.slide` to find all `div` elements with class value `slide`
    - `div li` to find all `li` elements within `div` elements
    - `div.slide > h1` to find all `h1` elements which are children of `div` elements with class value `slide`
    - `a[href^='http']` to find all `a` elements with an `href` attribute whose value starts with `http`

6.20. Adding elements

- the button Add li element is defined as follows:

  ```
  <button id="addButton">Add li element</button>
  ```

- with `click` event handler is assigned as follows:

  ```
  document.getElementById("addButton").onclick = addElement;
  ```

- the `ul` element on this slide is identified by `id="target1"`
- function `addElement` is defined as follows (in `client.js`):

  ```
  function addElement() {
    var elem = document.getElementById("target1");
    var node = document.createElement("li");
    var text = document.createTextNode("Hello");
    node.appendChild(text);
    elem.appendChild(node);
  }
  ```

  which appends a new `li` element to the identified `ul` element
- `createElement` is a *method* which creates an element with the given name
- `createTextNode` is a *method* which creates a text node with the given value
- `appendChild` is a *method* of a `node`; it appends the given node to the list of the node's children

6.21. Deleting elements

- the button Delete ul element is defined as follows:

  ```
  <button id="deleteButton">Delete ul element</button>
  ```

- with `click` event handler is assigned as follows:

  ```
  document.getElementById("deleteButton").onclick = deleteElement;
  ```

- the `ul` element on this slide is identified by `id="target2"`
- function `deleteElement` is defined as follows (in `client.js`):

  ```
  function deleteElement() {
    var elem = document.getElementById("target2");
    elem.parentNode.removeChild(elem);
  }
  ```

  which deletes the identified `ul` element

- removeChild is a *method* of a node; it removes the given node from the list of the node's children

6.22. Using jQuery

- jQuery is a popular JavaScript library which simplifies DOM operations
- it also takes care of differences among browsers
- the jQuery file `jquery-3.3.1.min.js` is referenced by these pages
- instead of using the `deleteElement` function, we could use

  ```
  function() {$('#target2').remove();}
  ```

- jQuery defines the object/method named `$`
  - which can take a *CSS selector* as an argument
  - and returns the collection of elements selected
- the `remove` method deletes the elements on which it is invoked

6.23. Using jQuery to add elements

- instead of using the `addElement` function, we could use

  ```
  function() {$('#target1').append($('<li />', {'text':'Hello'}));}
  ```

- the `append` method adds a new last child to elements on which it is invoked
- the function `$` creates a new element when passed a string representing an empty element as first argument
- the second argument is an *object* comprising property-value pairs:
  - the `text` property is interpreted as the textual contents of the element
  - other properties are interpreted as attribute names

6.24. Finding elements by name (jQuery)

- using jQuery, the example of finding headings could be done as follows:

  ```
   var ul = $('#headingList').append($('<ul />'));
   $('h1').each(function() {
                  ul.append($('<li />', {'text': $(this).text()}));
                });
  ```

- `each` iterates over a set of elements; the function it takes as argument is called for each element
- `this` returns the element on which the function is called
- `text` returns the textual contents of an element

6.25. DOM and XML

- JavaScript can use DOM objects, properties and methods to read and navigate through an XML document like `rss-fragment.xml`
- recall its structure is as follows

  ```
  <rss>
    <channel>
      <title> ... </title>
        ...
      <item>
        <title> ... </title>
  ```

```
      <description> ... </description>
      <link> ... </link>
      <pubDate> ... </pubDate>
    </item>
      ...
    <item>
      <title> ... </title>
      <description> ... </description>
      <link> ... </link>
      <pubDate> ... </pubDate>
    </item>
  <channel>
</rss>
```

- uses the XML parser built into the browser to construct a DOM tree

6.26. jQuery method to load an XML file

```
$.get("rss-fragment.xml",
      function( xml ) {
        ...
      },
      "xml"
    );
```

- this uses the `get` method of jQuery
- allows you to retrieve an XML file *from the same domain as the loaded page* (same origin policy)
- the method takes (at least) 3 arguments:
    - the first is the URL
    - the second is a function to be called on successful retrieval; it is passed the data retrieved (the root of the DOM tree)
    - the third is the type of the data retrieved (XML)

6.27. Retrieving RSS item titles

- say we want to return a list of RSS item titles when the following button is clicked:
  List item titles
- the `id` of the above button is `listButton`
- in `client.js` we bind a function name to the click event for the button:

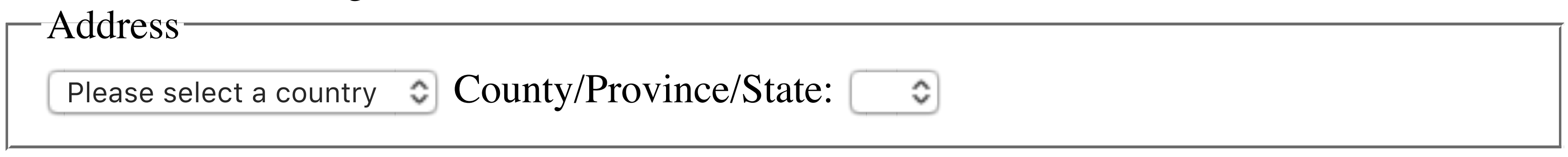
  ```
  document.getElementById("listButton").onclick = findItemTitles;
  ```

- there is an empty `div` element below, with `id="TitleList"`
- `findItemTitles` simply executes the code on the previous slide where the body of the function is:

  ```
  var titles = xml.querySelectorAll("item > title")
  var output = "<ul>";
  for ( i = 0; i < titles.length; i++ )
    output = output + "<li>" + titles[i].firstChild.nodeValue + "</li>";
  document.getElementById("TitleList").innerHTML = output + "</ul>";
  ```

6.28. Retrieving JSON data

- consider the following form:



Address

Please select a country County/Province/State:

- the code for the form is as follows:

```
<form>
  <fieldset>
    <legend>Address</legend>
    <select id='addressCountry'>
      <option value='na'>Please select a country</option>
      <option value='ca'>Canada</option>
      <option value='gb'>United Kingdom</option>
      <option value='us'>United States</option>
    </select>
    <label for='addressState'>County/Province/State:</label>
    <select id='addressState'></select>
  </fieldset>
</form>
```

- say we want to populate the drop-down list of counties/provinces/states by retrieving a list for the selected country from the server
- we also want to overwrite County/Province/State with the term appropriate for the country selected
- (This example is taken from the book "Web Development with jQuery" by Richard York, Wrox Press, 2015.)

6.29. JSON country data

- there are JSON files for each country: `ca.json`, `gb.json` and `us.json`
- the one for the UK (`gb.json`) looks as follows:

```
{
    "name" : "United Kingdom",
    "iso2" : "GB",
    "iso3" : "GBR",
    "label" : "County",
    "states" : {
        "0"   : " ",
        "794" : "Angus",
        ...
        "988" : "York"
    }
}
```

- `label` gives the correct term for County/Province/State
- `states` will be used to populate the drop-down list

6.30. Code for JSON retrieval (1)

- the code is as follows:

```
$('#addressCountry').change(
  function() {
    // Remove all of the options
    $('#addressState').empty();
    if (this.value === 'na') return;
    $.getJSON(
      this.value + '.json',
      function(json) {
        // Change the label
        $('label[for="addressState"]').text(
          json.label + ':'
        );
        ... see next slide
      }
    );
  }
);
```

- `change` (rather than `click`) event is needed for `select` in Chrome and Safari
- jQuery provides a `getJSON` function which retrieves a JSON file from the server
  - first argument is the name of the file
  - second argument is a function to be called on success and passed the JSON data

6.31. Code for JSON retrieval (2)

- the code for setting the options is as follows:

```
        // Set the options ...
        $.each(
          json.states,
          function(id, state) {
            $('#addressState').append(
              $('<option/>')
                .attr('value', id)
                .text(state)
            );
          }
        );
```

- `each` iterates over each key/value pair of the `states`
- for each one, the function is called and passed the key and value (`id` and `state`)
- a new `option` is added each time, with its `value` attribute set to the `id` of the state and its text set to the value of `state`

6.32. Exercises

1. Implement the following functions:
   - Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature using the calculation C= 5.0/9.0 * (F-32)
   - Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature using the calculation F = 9.0/5.0 * C + 32
2. Use the functions from (1) to write a script that enables the user to enter either a Fahrenheit temperature and display the Celsius equivalent or enter a Celsius temperature and display the Fahrenheit equivalent. You can either write two new functions that call your functions from (1)

or modify those functions instead. Your HTML document should contain a form with two buttons and two input fields. The one button initiates the conversion from Fahrenheit to Celsius, while the other initiates the conversion from Celsius to Fahrenheit. The input fields hold the two temperatures: one as input, the other as output. For simplicity, call the functions using `onClick` attributes. (see slide 15 and slide 16.)
3. Now modify the code in (2) so that it does not use `onClick` attributes but instead binds events to the buttons. Remember to use `window.onload` to delay the bindings until after the page has loaded (as in slide 13).
4. Reproduce the code to output information from the RSS document fragment as on slide 27 and slide 28, but outputting item descriptions rather than titles. For this, you will need to save a copy of the RSS document in your own web space (see the intranet for instructions). Your HTML page will also need to be there and you will need to access it over the web, using `titan.dcs.bbk.ac.uk/~...`, where `...` is your username. Your HTML page will also need to reference the jQuery library, either remotely or by saving a copy in your web space. The version I use is here.

6.33. Links to more information

- `www.webteacher.com/javascript/`
  JavaScript Tutorial for the Total Non-Programmer
- `www.w3schools.com/js/default.asp`
  JavaScript Tutorial
- `www.w3.org/DOM/`
  W3C's DOM web page
- `www.w3.org/TR/2000/WD-DOM-Level-1-20000929/ecma-script-language-binding.html`
  W3C DOM objects, properties and methods in JavaScript
- `www.w3schools.com/dom/default.asp`
  DOM Tutorial
- Introduction to events (MDN web docs)
- a good introductory book on jQuery is: Beginning jQuery, by Jack Franklin, Apress, 2013
- a more comprehensive book on jQuery is: Web Development with jQuery, by Richard York, Wrox Press, 2015

There are many books devoted to Javascript and/or jQuery. DOM is covered in Chapter 7 of [Moller and Schwartzbach] and Chapter 8 of [Jacobs]. See above for books on jQuery.