JavaScript and the Document Object Model

This workbook provides a basic introduction to the Document Object Model (the DOM) and how to manipulate it using JavaScript. It assumes that you have already completed the Introduction to JavaScript workbook. This workbook follows the same conventions as the previous workbook, it includes explanations, code descriptions and code examples, which you should read. It also includes activity sections which ask you to carry out a set of actions, for example to create or modify some code. The activities reinforce what you have read, but also give you vital coding practice. The workbook is design to be read in sequence, if you skip parts it may make it harder to understand later parts.

Whilst the workbook includes everything you need to cover, you may find it useful to refer to other sources of information. The w3Schools website has a course on JavaScript which includes a section on the DOM. Our library also has books on JavaScript.

The Document Object Model

The DOM is not part of HTML and it is not part of JavaScript, the DOM is part of the browser. The DOM provides two useful things – a model of the HTML page (the DOM tree) and a set of methods and properties which can be used to access and update objects in the DOM tree. The DOM tree is held in the browsers memory and we interact with the browser program to manipulate it.

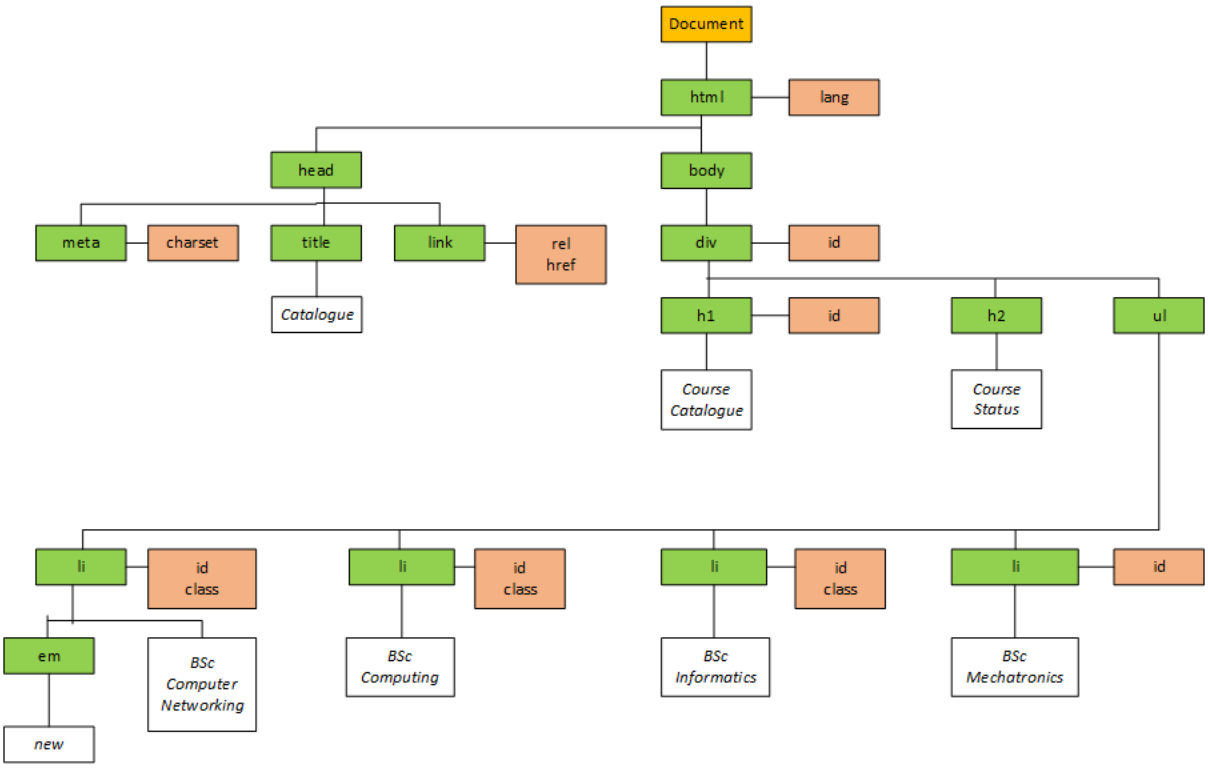
The Dom tree – the model of the HTML page – consists of four main types of node, the document node, element nodes, attribute nodes and text nodes. Consider this simple HTML page:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Catalogue</title>

    <link rel="stylesheet" href="domcss.css">
  </head>

  <body>
    <div id="page">
      <h1 id="header">Course Catalogue</h1>
      <h2>Course Status</h2>
      <ul>
        <li id="one" class="running"><em>new</em> BSc Computer
Networking</li>
        <li id="two" class="running">BSc Computing</li>
        <li id="three" class="running">BSc Informatics</li>
        <li id="four">BSc Mechatronics</li>
      </ul>
    </div>
  </body>
</html>
```

We can picture the DOM tree as shown below.



The DOM tree is rooted at the Document Node (yellow), access to all the nodes in the DOM tree start from this point.

The Element Nodes (green) represent the different items of HTML markup that define the page.

Some of the Element Nodes have attributes associated with them, these are held in Attribute Nodes (Pink) as part of the Element Node (they are not *children* of that Node).

Content is held in Text Nodes (white). A Text Node cannot contain any further nodes, they are the leaves of the tree.

Often when we are referring to the DOM tree we use terms which treat it as if it was a family tree, such as *parent*, *child* and *sibling*.

In the DOM tree above, the 'html' Element Node is the *parent* of the 'head' Element Node and the 'body' element Node. The 'head' Element Node and the 'body' Element Node are *children* of the 'html' Element Node. The 'head' Element Node and the 'body' Element Node are *siblings*.

Remember, Attribute Nodes are not *children* of their associated Element Node, they are part of that Element Node.

The nodes also have an order which corresponds to the order they appear in the DOM tree (which corresponds to the order they appear in in the HTML document). So, the 'h1' Element Node is the first child of the 'div' Element Node, the 'h2' Element Node is the second child of the 'div' Element Node, and the 'ul' Element Node is the third and last child of the 'div' Element Node.

Activity

1. Create a new HTML file, dom1.htm, containing the above code.
2. Create a new css, domcss.css, containing the following CSS rules:

```
body {
```

```
  background-color: hsl(0, 0%, 0%);
  font-family: sans-serif;
  margin: 0;
  padding: 0;}

#page {
  background-color: hsl(290, 5%, 50%);
  margin: 0 auto 0 auto;}

h1 {
  color: hsl(197, 50%, 90%);
  margin: 0 auto 0 auto;
  padding: 30px 10px 20px 10px;
  text-shadow: 2px 2px 1px hsl(290, 5%, 35%);}

h2 {
  color: hsl(176, 50%, 90%);
  margin: 0 0 10px 0;
  padding: 0px 10px 20px 10px;
  text-shadow: 2px 2px 1px hsl(290, 5%, 35%);}

ul {
  background-color: hsl(290, 5%, 50%);
  border: none;
  padding: 0;
  margin: 0;}

li {
  background-color: hsl(290, 5%, 80%);
  color: hsl(360, 100%, 100%);
  border-top: 1px solid hsl(290, 5%, 90%);
  border-bottom: 1px solid hsl(290, 5%, 70%);
  list-style-type: none;
  text-shadow: 2px 2px 1px hsl(290, 5%, 70%);
  padding-left: 1em;
  padding-top: 10px;
  padding-bottom: 10px;}

.running {
  background-color: hsl(155, 60%, 50%);
  text-shadow: 2px 2px 1px hsl(155, 60%, 40%);
  border-top: 1px solid hsl(155, 60%, 60%);
  border-bottom: 1px solid hsl(155, 60%, 40%);}

.full {
  background-color: hsl(12, 60%, 50%);
  text-shadow: 2px 2px 1px hsl(12, 60%, 40%);
  border-top: 1px solid hsl(12, 60%, 60%);
  border-bottom: 1px solid hsl(12, 60%, 40%);}
```

3. Upload both pages to the server and view the page in a browser.
4. Create a new JavaScript file, domexample.js
5. Add a link to domexample.js from dom1.htm

We will manipulate the DOM tree for dom1.htm in various ways to illustrate and understand the different methods for accessing and manipulating its Elements.

Selecting an individual Element Node

`getElementById()`

We can use this method to select a specific named Element Node – remember a particular id can only occur once in an HTML document.

```
function manipulateDOM1(){
    var el = document.getElementById('one');
    el.textContent = 'BSc Big Data';
}
```

We have used this previously in the Introduction to JavaScript workbook, so it should be familiar. Note that we are actually manipulating the DOM tree in two ways in this example. First, we are using `getElementById()` to access an Element Node in the DOM tree (we are storing a pointer to the node in the variable `el`). Second, we are changing the value of the `textContent` property of that Element Node.

`querySelector()`

We can use this method to select an Element Node that matches a specified CSS selector. Note that this method only returns the first Element Node that matches the CSS selector. If we want to select more than one, we need to use the `querySelectorAll()` method discussed further on.

```
function manipulateDOM2(){
    var el = document.querySelector('li.running');
    el.className = 'full';
}
```

So, this will access the first Element Node in the DOM tree that is identified by the CSS selector `li.running`, that is the first `li` Element Node that has `'class=running'`. We are then changing the `className` property of that Element Node from `running` to `full`, so that Element Node now has `'class=full'`. As our CSS specifies different styling for `.full`, the style of this element will be updated in the browser. So, we can change the style of Node Elements by manipulating their class using the `className` property.

Activity

6. Examine the `manipultateDOM1()` and `manipulateDOM2()` functions above and predict the change they will cause to the display of the HTML file.
7. Add the two functions to domexample.js.
8. Check to see if your predications were correct.

Selecting multiple Element Nodes

The two methods in the previous section both return a single Element Node, the methods in this section can return multiple Node Elements. The Node Elements are return in a data structure called a *NodeList*. A NodeList works in much the same way as an array, but is in fact a different type of data structure, a *collection*. A NodeList has properties, such as `length` and we can access individual items in the NodeList using an index number, where the items are indexed starting from zero, the same as an array. These methods will return a NodeList even if they are only returning one item.

`getElementsByClassName()`

We can use this method to select all the Element Nodes that have the specified class name.

```
function manipulateDOM3(){
    var els = document.getElementsByClassName('running');
    var noOfElements = els.length;

    if (noOfElements >= 1) {
        var el = els[noOfElements - 1];
        el.className = 'full';
    }
}
```

This will create a NodeList containing all the Element Nodes which have `class='running'`. If the NodeList contains at least one item, then the class of the last item is changed to `'full'`.

`getElementsByTagName()`

We can use this method to select all the Element Nodes that have the specified Tag Name (in other words, a specified HTML Tag).

```
function manipulateDOM4(){
    var els = document.getElementsByTagName('li');
    var noOfElements = els.length;

    if (noOfElements >= 1) {
        var el = els[noOfElements - 1];
        el.className = 'full';
    }
}
```

This will create a NodeList containing all the Element Nodes which have the Tag Name `li`. If the NodeList contains at least one item, then the class of the last item is changed to `'full'`.

`querySelectorAll()`

We can use this method to select all the Element Nodes that match a specified CSS selector.

```
function manipulateDOM5(){
    var els = document.querySelectorAll('li.running');
    var noOfElements = els.length;

    for (var i = 0; i < noOfElements; i++ ) {
        els[i].className = 'full';
    }
}
```

This will create a NodeList containing all the Element Nodes which are identified by the CSS selector `li.running`, that is all the `li` Element Nodes that have '`class=running`'. The class of each of the items in the NodeList is changed to `'full'`.

Activity

9. Examine the `manipultateDOM3()`, `manipultateDOM4()` and `manipulateDOM5()` functions above and predict the change they will cause to the display of the HTML file.
10. Add the functions to domexample.js.
11. Check to see if your predications were correct.
12. Consider the `manipultateDOM6()` and `manipulateDOM7()` functions below:

```
function manipulateDOM6(){
    var el = document.querySelector('li.running');
    var els = document.querySelectorAll('li.running');

    /* set class of first element to full */
    el.className = 'full';

    /* set class of first element to full */
    els[0].className = 'full';
}


function manipulateDOM7(){
    var el = document.querySelector('li.running');

    /* set class of first element to full */
    el.className = 'full';

    var els = document.querySelectorAll('li.running');

    /* set class of first element to full */
    els[0].className = 'full';
}
```

They both contain the same lines of code, but do they do the same thing?

13. Add the two functions to domexample.js.
14. Check to see if your predication was correct.

The `manipultateDOM6()` and `manipulateDOM7()` functions produce different results because the JavaScript is executed one line at a time, if we manipulate the DOM tree, then these changes can affect the lines of code that follow. In `manipultateDOM6()` we set the values of `el` and `els` before we made any changes to the DOM tree. The Element Node in `el` and the first Element Node in the `els` NodeList are both the `li` with `id='one'`. We set its class value to `'full'` twice.

In `manipulateDOM7()` we didn't set the value of `els` until after we had made a change to the DOM tree. Because we had already changed the class of the first element node (in this case the `li` with `id='one'`) from `'running'` to `'full'`, this element node wasn't include in the `els` NodeList. The first element node in the `els` NodeList is the `li` with the `id='two'`, so we end up setting this Element Nodes class value as well.

This is something we need to be mindful of. There is another slight twist to this, which is the difference between a live NodeList and a static NodeList.

Activity

15. Consider the `manipultateDOM8()` function below:

```
function manipulateDOM8(){
  var els = document.getElementsByClassName('running');

    els[1].textContent='BA Social Media Studies';

    /*
    var el = document.getElementById('two');
    el.className = '';
    */

    els[1].className='full';
}
```

    As it stands, this function creates a NodeList of Element Nodes with `class='running'`, then it sets the `textContent` of the second item in the NodeList to `'BA Social Media Studies'` and its `className` to `'full'`

16. Add the function to domexample.js and verify its behaviour.
17. Consider, what will happen if we uncomment the code in the middle? Uncomment the code and verify its behaviour.

The result of uncommenting the code may seem rather perplexing. The contents of the NodeList have changed between setting the `textContent` and setting the `className`, even though we didn't manipulate it in any way. This is because `getElementsbyClassName()` and `getElementsByTagName()` both return live NodeLists. A live NodeList is automatically updated when changes are made to the DOM tree. Changing the `className` of the Element Node with `id='two'` meant that it no longer met the criteria (`class = 'running'`) for the `els` NodeList, so it was removed. This means that the second item in the `els` NodeList is now a different Element Node when the `className` is set to `'full'`.

Note that `querySelectorAll()` returns a static NodeList, its contents are not updated when changes are made to the DOM tree.

Moving between Element Nodes

Moving between Element Nodes is complicated by the fact that different browsers treat whitespace characters (such as spaces or carriage returns) differently. Most treat them as Text Nodes, but IE does not. We can address this is several ways; we could strip out the whitespace characters, we could avoid using these functions, or we could make use of a JavaScript library (such as jQuery which we will look at in a later workbook) to deal with the issue.

For now we will remove the whitespace characters to that we can explore how the functions work:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Catalogue</title>
```

```
    <link rel="stylesheet" href="domcss.css">
  </head>

  <body>
    <div id="page">
      <h1 id="header">Course Catalogue</h1>
      <h2>Course Status</h2>
      <ul><li id="one" class="running"><em>new</em> BSc Computer
Networking</li><li id="two" class="running">BSc Computing</li><li
id="three" class="running">BSc Informatics</li><li id="four">BSc
Mechatronics</li></ul>
    </div>
  </body>
</html>
```

We can immediately see how less readable our HTML is.

Activity

18. Create a new HTML file, dom2.htm, containing the above code.
19. Add a link to domcss.css and domexample.js from dom2.htm
20. Upload dom2.htm to the server and view the page in a browser. It should look no different.
21. Add the following rules to the CSS:

    ul border-radius: 0px;

    li border-radius: inherit;

22. Add the following class to the CSS:

    .rounded {border-radius: 10px;}

23. View the HTML page in a browser, it should look no different.

`parentNode`

We can use this property to access the Element Node for the parent of the current node.

```
function manipulateDOM9(){
    var el = document.getElementById('one');
    var parel = el.parentNode;
    parel.className = 'rounded';
}
```

We have accessed a Node Element by its id, then accessed its parent Element Node using the `.parentNode` property. We have then set the `class` attribute of that parent node to `'rounded'`. This means that any style rules in the CSS that apply to the `.rounded` class are now applied to the parent node.

Activity

24. Examine the `manipulateDOM9()` function above.
25. Add the function to domexample.js.

26. Compare the HTML page before and after running `manipulateDOM9()`.

`previousSibling`

`nextSibling`

We can use these properties to access the Element Nodes for the siblings of the current node. One example of siblings in our example DOM tree are the list item Element Nodes, these are siblings of each other.

```
function manipulateDOM10(){
    var el = document.getElementById('two');
    var preSib = el.previousSibling;
    var nexSib = el.nextSibling;

    if (preSib !== null) {
        preSib.className = 'full';
    }
    if (nexSib) {
        nexSib.className = 'full';
    }
}
```

We have accessed a Node Element by its id, then accessed its previous sibling Element Node using the `.previousSibling` property, and its next sibling Element Node using the `.nextSibling` property. In our example, the previous sibling of the Element Node with `id = 'two'` is the Element Node with `id = 'one'` and the next sibling is the Element Node with `id = 'three'`, but these ids are arbitrary, they have nothing to do with indicating previous or next sibling, this is determined purely from the DOM tree.

If there is no previous or next sibling, then `null` is returned when we access the property. `ManipulateDOM10()` shows two different ways of testing for a null return value. With preSib we have used `!==` the "strict not equal to" operator in the condition. With nexSib it doesn't look like there actually is a condition. This is in fact just a shorthand for "nexSib is true". This may appear incorrect, nexSib will not contain the Boolean value true, so we might expect the condition to evaluate to false even when there was a next sibling (and nexSib contains a pointer to that Element Node). However, in addition to the Boolean true and false values, JavaScript has Truthy and Falsy values, that is values which are treated as true and false, even though they aren't really.

Falsy Values

Value	**Description**
`var ourVariable = false;`	Boolean false
`var ourVariable = 0;`	The number zero
`var ourVariable = '';`	NaN (Not a Number)
`var ourVariable = 10/'string';`	Empty value
`var ourVariable;`	No value assigned

Pretty much everything else is truthy, including…

Truthy Values

Value	Description
`var ourVariable = true;`	Boolean false
`var ourVariable = 1;`	Numbers aside from zero
`var ourVariable = 'string';`	Strings with content
`var ourVariable = 10/5;`	Number calculations
`var ourVariable = 'true';`	True written as a string
`var ourVariable = '0';`	Zero written as a string
`var ourVariable = 'false';`	false written as a string

So, in `manipulateDOM10()` the nexSib condition check evaluates to true if there is a next sibling because it is truthy.

Activity

27. Examine the `manipultateDOM10()` function and predict the change it will cause to the display of the HTML file.
28. Add the function to domexample.js.
29. Check to see if your predications were correct.
30. Change `manipultateDOM10()` so that it selects the Element Node with `id='three'` instead of `id='two'`
31. Predict the change this will cause to the display of the HTML file.
32. Check to see if your predications were correct.

The `className` of the list item containing 'BSc Mechatronics' (the `nextSibling` of the Element Node with `id='three')` has been set to `'full'` even though it didn't previously have a class attribute defined in the HTML file. We can add a class to an Element Node using the `className` property. We did this previously in the `parentNode` example, but you may not have noticed.

`firstChild`

`lastChild`

We can use these properties to access the Element Nodes for the children of the current node. The child relationship is the reverse of the parent relationship.

```
function manipulateDOM11(){
    var els = document.getElementsByTagName('ul');

    if (els.length >= 1) {
```

```
        var firChi = els[0].firstChild;
        var lasChi = els[0].lastChild;

        if (firChi) {
            firChi.className = 'full';
        }
        if (lasChi) {
            lasChi.className = 'full';
        }
    }
}
```

We have created a NodeList containing all the `ul` Element Nodes. In our DOM tree there is only one, but it will still be returned as a NodeList. Then, if there is at least one Element Node in the NodeList (`els`), we set the className of the first and last child Element Node of the first Element Node (`els[0]`), to `'full'`.

Activity

33. Examine the `manipulateDOM11()` function and predict the change it will cause to the display of the HTML file.
34. Add the function to domexample.js.
35. Check to see if your predications were correct.
36. Change `manipulateDOM11()` so that it accesses the children Element Nodes of the first `ul` Element Node in the DOM tree and changes the class of any child Element Node that is `'running'` to `'full'` and any that is not `'running'` (there is no `class` attribute, `className === ''`) to `'running'`.

Accessing and updating a Text Node

There are two main approaches we can take when accessing and updating Text Nodes. We can work directly with the Text Node, or we can work via its containing Element Node.

nodeValue

This property allows us to select and amend the contents of a Text Node.

```
function manipulateDOM12(){
    var el = document.getElementById('four');
    var elTxt = el.firstChild.nodeValue;

    elTxt = elTxt.replace('Mechatronics', 'Multimedia Studies');
    el.firstChild.nodeValue = elTxt;

    document.getElementById('one').firstChild.nextSibling.nodeValue
= ' BSc Games Design';
```

```
}
```

In `manipulateDOM12()` we are changing the content of two Text Nodes. First we are working with the Text Node contained by the Element Node with `id='four'`. We are using the String object `replace()` method which replaces the first occurrence of the target string (in this case `'Mechatronics'`) with a new string (in this case `'Multimedia Studies'`). We are only changing part of the Text Node content. With the second Text Node example, we are accessing and changing the Text Node content in a single statement. If you look at it carefully you will see that we are using one method and three properties to access the Text Node content.

Activity

37. Examine the `manipultateDOM12()` function and predict the changes it will cause to the display of the HTML file.
38. Add the function to domexample.js.
39. Check to see if your predications were correct.
40. Add a single statement to `manipultateDOM11()` that changes the Text Node contained by the Element Node with `id='two'` to `'BSc Artificial Intelligence'`.

As we can see in the above examples, it can be complicated to access Text Nodes where there are sibling Element Nodes, or where we don't know whether there will be. For this reason we will often work with the containing Element Node instead.

`textContent`

This property allows us to access and manipulate text content via its containing Element Node.

```
function manipulateDOM13(){
    var el = document.getElementById('four');
    var elTxt = el.textContent;

    elTxt = elTxt.replace('Mechatronics', 'Multimedia Studies');
    el.textContent = elTxt;
}
```

In this example we are replicating the effect of the first part of `manipulateDOM12()`, but we are working with the containing Element Node rather than the Text Node.

Activity

41. Examine the `manipultateDOM13()` function and predict the changes it will cause to the display of the HTML file.
42. Add the function to domexample.js.
43. Check to see if your predications were correct.
44. Add the following code to `manipulateDOM13()`

```
el = document.getElementById('one');
el.textContent = ' BSc Games Design';
```

45. Predict the changes it will cause to the display of the HTML file, will it replicate the effect of the second part of `manipulateDOM12()`?.
46. Check to see if your predications were correct.

As we can see, this does not exactly replicate the effect of the second part of `manipulateDOM12()`. All of the text content below the Element Node has been replaced, including the text content in the child `em` Element Node. This can have quite a dramatic effect on the DOM tree, consider for example:

```
var els = document.getElementsByTagName('body');
els[0].textContent = '';
```

Working with HTML content

In addition to amending the text content of the DOM Tree, we may also want to amend the actual HTML itself.

`innerHTML`

We have encountered innerHTML previously in the Introduction to JavaScript Workbook. It basically uses strings which can contain HTML markup.

```
function manipulateDOM14(){
    var courseUpdated = '<b> Updated!</b>'
    var el = document.getElementById('four');
    var elTxt = el.textContent;

    elTxt = elTxt + courseUpdated;
    el.innerHTML = elTxt;
}
```

In this example we are appending a string which includes some HTML markup to existing content contained by an Element Node. We are then overwriting the old content with this new content. Because we are using the `.innerHTML` property and not the `.textContent` property, the HTML markup will be interpreted as HTML markup, rather than just text content.

Activity

47. Examine the `manipultateDOM14()` function and predict the changes it will cause to the display of the HTML file.
48. Add the function to domexample.js.

49. Check to see if your predications were correct.

We can also use the `.innerHTML` property to remove Element Nodes from the DOM tree.

```
function manipulateDOM(){
    var el = document.getElementById('one').firstChild;
    el.innerHTML = '';
}
```

Whilst the .innerHTML property is simple to use and powerful, there can be security risks associated with using it. For this reason, we may prefer the other approaches below.

`createElement()`

We can use this method to create new Element Nodes. These Element Nodes are created without content and are not initially connected to the DOM tree. We need to add the Text Node and link the new Element Node into the DOM tree ourselves.

`createTextNode()`

This method creates a new Text Node. Again the new Text Node is not initially connected to the DOM tree, we must connect it ourselves.

`appendChild()`

This method allows us to connect Text Nodes to Element Nodes and Element Nodes to other Element Nodes. This is how we connect our New Element and Text Nodes to the DOM tree.

```
function manipulateDOM15(){
    var newEl = document.createElement('li');
    var newTxt = document.createTextNode('BSc Theoretical
Robotics');

    newEl.appendChild(newTxt);
    var loc = document.getElementsByTagName('ul')[0];
    loc.appendChild(newEl);
}
```

This example create a new `li` Element Node and a new Text Node containing the string `'BSc Theoretical Robotics'`, it then attaches the new Text Node to the new Element Node. It the attaches the new Element Node to the `ul` Element Node (technically to the first `ul` Element Node, but we only have one).

Activity

50. Examine the `manipultateDOM15()` function and predict the changes it will cause to the display of the HTML file.
51. Add the function to domexample.js.
52. Check to see if your predications were correct.

Note that the new Element Node is attached as the last child when we use `appendChild()`. If we want to add it somewhere else we can use the `insertBefore()` method, so for example if we replaced `loc.appendChild(newEl);` with `loc.insertBefore(newEl, loc.firstChild);` in `manipulateDOM15()`,the new Element Node would be placed before the first child of the `ul` Element Node.

`removeChild()`

This method allows us to remove Element Nodes from the DOM tree. In order to use it we need to have a pointer to the Element Node and a pointer to the Element Nodes parent Element Node.

```
function manipulateDOM16(){
    var delEl = document.getElementsByTagName('li')[2];
    var parDelEl = delEl.parentNode;

    parDelEl.removeChild(delEl);
}
```

In this example we are deleting the third list item from the DOM tree.

Activity

53. Create a new `manipultateDOM()` function to do the following
54. Create a new list item Element Node. The list item content consists of the string 'new' which is emphasised (`em` Tag) and the string `' BSc Theoretical Robotics'` which is not.
55. Add the new list Item Element node into the list of courses before the first course that is not `'running'` (there is no `class` attribute, `className === ''`). Do not make any assumptions about which Element Node it might be (i.e. do not rely on position or `id` to identify it, check the attribute). If there is no non-running course, the New Element Node should be added at the end of the list of courses. If there are no courses in the list, then the new Element Node is added to the list.

    You may find the `break` keyword useful here. It immediately exits the current loop (`for`, `while` or `do while`) and continues with the next statement after the loop. For example

```
for (i = 0; i < arrayLength; i++ ) {
    …
    if (updated === true){
        break;
```

```
        }
        …
    }
    // will break to here if updated === true
```

56. Check that the function works for the edge cases: when the first non-running course is the first course in the list; there is no non-running course; there are no courses in the list.
57. Change the attribute of the new Element Node to show that the course is running
58. Change the attribute of the Element Node with `id='two'` to indicate that it is not running (`class=''`)
59. Remove all `li` Element Nodes which contain courses which are not running. Do not make any assumptions about which Element Node/s it might be (i.e. do not rely on position or id to identify it, check the attribute).
60. Check that the function works correctly regardless of how many non-running courses there are.
61. Check that the function works for the edge cases: first course not running; last courses not running; when none of the courses on the list are running, when there are no courses on the list.

We should be in the habit of testing all our code as we develop it and of paying particular attention to edge cases – the extreme cases. Often we will find that our code works perfectly well for typical cases, but behaves oddly at the extremes. These often need to be trapped (recognised) and dealt with as special cases.

Accessing and updating Attribute Node values

We have already performed some access and manipulation of Attribute Nodes using the `className` property, but there are additional properties and methods we haven't used.

`hasAttribute()`

`getAttribute()`

The `hasAttribute()` method allows us to check if an Element Node has a specific attribute and the `getAttribute()` method allows us to retrieve the value of an attribute.

```
function manipulateDOM17(){
    var el = document.getElementsByTagName('li')[2];
    var revClass = ' Has no class attribute';

    if (el.hasAttribute('class')) {
        revClass = ' ' + el.getAttribute('class');
    }
```

```
        el.textContent = el.textContent + revClass;
    }
```

In this function we are using .hasAttribute() to check if the selected Node Element has an associated Attribute Node for the attribute 'class' (class='something'). If it does have a class attribute, then the value of the attribute is retrieved using the .getAttribute() method.

setAttribute()

We have already used the .className property to set, retrieve and change a class attribute. We can manipulate the id of an Element Node in the same way using the .id property (there are other properties too, but you can investigate these yourself). The .setAttribute() method allows us to set, retrieve and change any attribute value.

```
function manipulateDOM18(){
    var el = document.getElementsByTagName('link')[0];

    el.setAttribute('href', 'domcss2.css');
}
```

The example above would change the CSS associated with our HTML document. We could also use this for example to change the src of img Tags or the href of a Tags.

removeAttribute()

We could use .setAttribute() to remove an attribute, e.g. el.setAttribute('href', ''), but it may be clearer to use the .removeAttribute() method.

```
function manipulateDOM19(){
    var el = document.getElementsByTagName('li')[2];

    if (el.hasAttribute('class')) {
        el.removeAttribute('class');
    }
}
```

In this example we are checking to see if the Element Node has a class attribute and if it has, we are removing it. It is always good practice the check existence of an attribute before removing it, though it is not strictly necessary.

Activity

62. Create a new manipultateDOM() function to do the following
63. If there are less than two courses, do nothing. Otherwise, check how many courses are running. If it is zero, change the first two courses to be running. If it is an even number, do

nothing. If it is an odd number and there is at least one non-running course, change the first non-running course to be running. If it is an odd number and there are no non-running course, change the last running course to be non-running.

64. Check that your function works for edge cases.

Cross-Site Scripting Attacks

Earlier it was mentioned that the use of `.innerHTML` can introduce security risks into our site. We need to be cautious when we include content on our site that we have not produced ourselves, for example if we allow users to add comments or upload content or we draw data from other sites. It is possible that a malicious user could manipulate the DOM tree, access cookies or access session tokens which could allow them to access other users accounts.

In order to reduce the chances of this occurring we should validate user entered data. We should not allow untrusted users to submit HTML markup or JavaScript to the site. We should validate it when it is entered by the user and again at the server before storing the data in our database. When we retrieve it from the database and display in in an HTML page, we want to make sure that it is only ever treated as a text string, never as markup or instructions.

We should avoid placing user generated content between <script> Tags, in HTML comments, in Tag names, in attributes or in CSS values. All of these are possible exploits.

We should escape potentially problematic characters – that is convert them into version which will display correctly but will not be treated as markup or instructions. This should be done at the server-side. For example if the user content includes a < we should replace it with <

When we add untrusted content to our HTML page we should use `.textContent` or `.innerText` to add it as text rather than `.innerHTML` which would enable them to include HTML markup.

Essentially we want to do everything we can to make sure that untrusted user content only contains text and is only ever manipulated as text. Where possible on forms we should allow users to select from a set of options, rather than allow them to enter their own free content.