Node Navigation Basics

In this lesson, let's see some node navigation basics.

WE'LL COVER THE FOLLOWING

The concept of root nodes, parents, children, sibling nodes, text nodes, and attributes are very important because the navigation in the tree is based on these ideas.

The document tree is represented by the document JavaScript object. There are a number of document operations dedicated to navigation. Listing 6-3 (it is based on the markup shown in Listing 6-2) demonstrates a few methods of navigation.

Listing 6-3: Demonstrating methods of navigation

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Tree</title>
  <script>
    function logNavigation() {
      console.log('Children of the document')
      var children = document.childNodes;
      for (var i = 0; i < children.length; i++) {
        logNodeInfo(i, children[i]);
      }

      console.log('Children of <body>')
      var current = document.body.firstChild;
      var index = 0;
      while (current != null) {
        logNodeInfo(index, current);
```

```
          index++;
          current = current.nextSibling;
        }


        console.log('Children of <p>')
        children = document.getElementsByTagName('p')[0]
          .childNodes;
        for (var i = 0; i < children.length; i++) {
          logNodeInfo(i, children[i]);
        }

        console.log('Parent of <li>');
        var parent = document.getElementsByTagName('li')[0]
          .parentNode;
        logNodeInfo(0, parent);
      }

      function logNodeInfo(index, node) {
        var nodeVal = node.nodeValue == null
          ? '{empty}'
          : node.nodeValue.replace('\n', '{nl}');
        console.log('   ' + index + ': ('
          + node.nodeType + ') '
          + node.nodeName + ' | '
          + nodeVal);
      }
  </script>
</head>
<body onload="logNavigation()">
  <h1>Chapter <strong>#1</strong>: HTML Basics</h1>
  <!-- This is a comment -->
  <p class="firstPara">
    This is the first paragraph of this chapter
  </p>
  <ul>
    <li>Item #1</li>
    <li>Item #2</li>
  </ul>
</body>
</html>
```

The JavaScript code in this listing uses the `console.log()` method to add diagnostic messages to the console of the browser you use.

Using the `F12` key in Google Chrome or Edge, or `Ctrl+Shift+K` in Firefox will show you the developer tools, where you can find the console.

When displaying the page in Listing 6-3, the console will display this output:

console

```
Children of the document
  0: (10) html | {empty}
  1: (1) HTML | {empty}
Children of <body>
```

```
  1: (1) H1 | {empty}
  2: (3) #text | {nl}
  3: (8) #comment |  This is a comment
  4: (3) #text | {nl}
  5: (1) P | {empty}
  6: (3) #text | {nl}
  7: (1) UL | {empty}
  8: (3) #text | {nl}

Children of <p>
  0: (3) #text | {nl}    This is the first paragraph of this chapter

Parent of <li>
  0: (1) UL | {empty}
```

> **NOTE:** Earlier, in Exercise 6-2 you have already used the `document.write()` method. Because this method changes the document body itself, it would cause issues with navigating the page structure while altering it. Using `console.log()` avoids this potential problem.

The `<script>` section of the page (nested in `<head>`) contains only function definitions. The `onload` attribute of `<body>` is set to `“logNavigation()”`, so when the loading of the page has been completed, the `logNavigation()` method is immediately invoked; this carries out the navigation operations logged in the console output.

This method contains four chunks of operations that query the children of the document, of the `<body>` element, of the `<p>` element, and last, the parent of the `<li>` element, respectively. To display node details, it utilizes the `logNodeInfo()` method.

Before going on to examine the output, let’s see what `logNodeInfo()` does:

```
function logNodeInfo(index, node) {
  var nodeVal = node.nodeValue == null
    ? '{empty}'
    : node.nodeValue.replace('\n', '{nl}');
  console.log('  ' + index + ': ('
    + node.nodeType + ') '
    + node.nodeName + ' | '
    + nodeVal);
}
```

This method receives an `index` value, and a tree node object. It displays the `nodeType`, `nodeName`, and `nodeValue` properties of the node passed as an argument. Each node in the tree has a node type (`nodeType`), such as element, text, comment, `<!DOCTYPE>`, and a few others.

These are represented by integer values, as displayed in the output within parentheses.

Nodes have a name (`nodeName`) that varies according to the node type. Text node names are always set to `“#text”`.

Comments use the `“#comment”` constant as their name. HTML elements use the name of related markup tag, using uppercase letters.

To help reading the console output, the `logNodeInfo()` method replaces the empty nodeValue texts with the `“{empty}”` text, and new line characters within the text with the `“{nl}”` literal.

The children of the document node are listed with this code snippet:

```
console.log('Children of the document')
var children = document.childNodes;
for (var i = 0; i < children.length; i++) {
  logNodeInfo(i, children[i]);
}
```

This code is straightforward. The `document.childNodes` property retrieves the collection of child nodes, and the subsequent `for` loop iterates through them.

This snippet generates the following output:

console

```
Children of the document
  0: (10) html | {empty}
  1: (1) HTML | {empty}
```

The (`10`) integer refers to the `<!DOCTYPE html>` node. Its name, `“html”`, covers the document type (HTML5 document) and not the `<html>` node of the markup.

Just as shown in the document structure image, the document node has two children, the second one is the `<html>` node at `index 1` in the output. As you see, both nodes have empty values.

The same code could be used to display the children of `<body>`, however the code listing uses a different approach:

```
console.log('Children of <body>')
var current = document.body.firstChild;
var index = 0;
while (current != null) {
  logNodeInfo(index, current);
  index++;
  current = current.nextSibling;
}
```

It uses the `document.body` property, which represents the `<body>` node of the document.

To access the first child node of `<body>`, the code uses the `firstChild` property. The while loop displays the information about the current node, and after incrementing the node index, it uses the `nextSibling` method of the node, and not the one of the document, to get to the next child of the parent, which is the sibling of the previously touched child.

So, this loop gets the first child and iterates through that child's siblings.

The document structure image indicates that `<body>` has only three children, `<h1>`, `<p>`, and `<ul>`.

However, the console output shows much more:

console

```
Children of <body>
  0: (3) #text | {nl}
  1: (1) H1 | {empty}
  2: (3) #text | {nl}
  3: (8) #comment |  This is a comment
  4: (3) #text | {nl}
  5: (1) P | {empty}
  6: (3) #text | {nl}
  7: (1) UL | {empty}
  8: (3) #text | {nl}
```

Listing 6-3 includes an additional comment that is shown in the output at `index 3`. But where are the extra five text nodes coming from? As the output shows, they come from the line breaks between markup elements. For example, the text at `index 1` comes from the line break between `<body>` and `<h1>`:

```
<body onload="logNavigation()">
  <h1>Chapter <strong>#1</strong>: HTML Basics</h1>
```

When you remove the line break in the original markup and go on with `<h1>` right after the closing `“>”` of `<body>` (without spaces), the text node at `index 0` will disappear.

All other text nodes come from other line breaks within `<body>`.

A third approach is used to display the children of `<p>`:

```
console.log('Children of <p>')
children = document.getElementsByTagName('p')[0]
  .childNodes;
for (var i = 0; i < children.length; i++) {
  logNodeInfo(i, children[i]);
}
```

Now, the `document.getElementsByTagName()` method is used with `“p”` as its input argument. This call retrieves a collection of all `<p>` tags in the document. Using the `[0]` indexing tag gets the first item from this collection and evaluates the `childNodes` property of this item. It returns all child nodes of the first `<p>` element in the document. The subsequent for loop iterates through the child nodes.

> 📃**NOTE:** The code above assumes that there is at least one `<p>` tag in the markup, by using the `[0]` indexing expression. Of course, we know that it’s true, but in real life the code should check if there is any child before referring to the first one.

There is one very important thing the output of this code snippet indicates:

console

```
Children of <p>
  0: (3) #text | {nl}    This is the first paragraph of this chapter
```

The output does not display the class attribute of `<p>` which is set to "firstPara". Of course, the attribute node is the part of the document tree, but it is not a child node of `<p>`!

The last code segment is so easy to understand that it does not need extra explanation:

```
console.log('Parent of <li>');
var parent = document.getElementsByTagName('li')[0]
  .parentNode;
  logNodeInfo(0, parent);
}
```

The `parentNode` property of a node simply retrieves its parent node object.

As mentioned earlier, element attributes are document tree nodes, too. Because attributes are not children of their container node, they cannot be accessed by any child-related operations.

The document object has attribute-related navigation operations and properties, as Listing 6-4 demonstrates.

Listing 6-4: Attribute-related navigation operations and properties of the document object

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Tree</title>
  <style>
    .redcolor {
      color: red;
    }
  </style>
  <script>
    function logAttributes() {
      console.log('item1 attributes:');
      logElementAttrs('item1');

      console.log('item2 attributes:');
      logElementAttrs('item2');

      console.log('item3 attributes:');
      logElementAttrs('item3');

      console.log('item2 "class" attribute:');
      var attr = document.getElementById('item2')
        .getAttributeNode('class');
      logNodeInfo(0, attr);
    }
    function logElementAttrs(id) {
      var attrs = document.getElementById(id)
        .attributes;
      for (var i = 0; i < attrs.length; i++) {
        logNodeInfo(i, attrs[i]);
      }
    }

    function logNodeInfo(index, node) {
      console.log('  ' + index + ': ('
        + node.nodeType + ') '
        + node.nodeName + ' | '
        + node.nodeValue);
    }
  </script>
</head>
<body onload="logAttributes()">
  <p>This is an ordered list</p>
  <ol start="1">
    <li id="item1">Item #1</li>
    <li id="item2" class="redcolor">Item #2</li>
    <li id="item3" invalid="wrong"
        data-myAttr="custom">Item #3</li>
  </ol>
</body>
</html>
```

This markup displays a paragraph and an ordered list. The attached script works with the same approach as the one in Listing 6-3, but this time it traverses through element attributes. It displays all attributes of elements with `“item1”`, `“item2”`, and `“item3”` identifiers, respectively.

Additionally, the scripts display information on the class attribute of `“item2”`.

The `logNodeInfo()` method works exactly the same way as in the previous listing, except that it does not transform the input node’s value. The lion’s share of the work is done by `logElementAttrs()` that accepts an ID argument, and displays the attributes of the node specified by the identifier passed in.

The key is this code snippet:

```
var attrs = document.getElementById(id)
  .attributes;
```

The attributes property of an element, which is obtained by `document.getElementById()`, retrieves a collection of attribute nodes. The subsequent for loop in `logElementAttrs()` iterates through all attributes and displays their details.

When you display the page, it produces the following output:

console

```
item1 attributes:
  0: (2) id | item1
item2 attributes:
  0: (2) id | item2
  1: (2) class | redcolor
item3 attributes:
  0: (2) id | item3
  1: (2) invalid | wrong
  2: (2) data-myattr | custom
item2 "class" attribute:
  0: (2) class | redcolor
```

There are important things this output shows you. Attributes are nodes with `type 2`, as the `(2)` value in the output indicates. The attribute property retrieves all attributes, including valid (such as `id`, `class`), custom (`data-myattr`), and invalid (`invalid`) attributes as well.

The DOM does not care whether the browser can display the markup or not, it provides every detail of the document tree.

The last chunk of the script accesses the class attribute of `“item2”` directly:

```
var attr = document.getElementById('item2')
  .getAttributeNode('class');
```

This is the task of `getAttributeNode()`, which navigates to the attribute specified by its name as the argument of the function.

> 📜**NOTE:** The element object provides another method, `getAttribute()`, to access an attribute by its name. Although `getAttributeNode()` retrieves a node object, `getAttribute()` retrieves the attribute’s value (a string) only.

In the *next lesson*, we will access element content.