Dynamic Content

Dr. Patrick Young, Stanford University

CS193C

Accessing an Element

We have several methods available for accessing a particular element on our webpage. Use the most specific method available for efficiency. Searching for an element with a specific ID will be much faster using getElementById instead of using querySelector and passing in a CSS query looking for that exact same ID.

Using the getElementById Method

If we give an element an id we can access that element using the Document's getElementById method. Here we retrieve the <h1> tag with ID "topHeading" and set it to display in "red".

```
function change() {
  var heading = document.getElementById("topHeading");
  heading.style.backgroundColor = "red";
}
…
<body>
<h1 id="topHeading">Id Example</h1>
</body>
```

Using the getElementsByTagName Method

The Document's getElementsByTagName method retrieves a NodeList containing all the elements of a given tag type. In this example we retrieve an NodeList containing all three <p> tags and then set all of them to display in "red".

```
function change() {
  var pElems = document.getElementsByTagName("p");
  for(var i=0; i<pElems.length; i++) {
    pElems[i].style.backgroundColor = "red";
  }
}
…
<body>
<p>First Paragraph</p>
<p>Second Paragraph</p>
<p>Third Paragraph</p>
</body>
```

A NodeList acts very similar to an array and you may use the standard square bracket notation [] to access elements in the NodeList and the .length property to determine its length. However, many of the more powerful array methods are not available on it.

Using the getElementsByClassName Method

The Document's getElementsByClassName method retrieves a NodeArray containing all elements identified as belonging to a specific class.

```
function change() {
  var importantElems = document.getElementsByClassName("important");
  for(var i=0; i<importantElems.length; i++) {
    importantElems[i].style.backgroundColor = "red";
  }
}
…
<body>
<h1 class="important">Example</h1>
<p>First Paragraph</p>
<p>Second Paragraph</p>
<p class="important">Third Paragraph</p>
</body>
```

Using the querySelector Method

Pass a string containing a CSS query to Document's querySelector method and it will return the first element it finds which the selector.

```
function change() {
  var result = document.querySelector("p.important");
  result.style.backgroundColor = "red";
}
…
<body>
<h1 class="important">Example</h1>
<p>First Paragraph</p>
<p class="important">Second Paragraph</p>
<p class="important">Third Paragraph</p>
</body>
```

Using the querySelectorAll Method

Pass a string containing a CSS query to Document's querySelector method and it will return a NodeList of all elements which match the query.

```
function change() {
  var result = document.querySelector("p.important");
  result.style.backgroundColor = "red";
}
…
<body>
<h1 class="important">Example</h1>
<p>First Paragraph</p>
<p class="important">Second Paragraph</p>
<p class="important">Third Paragraph</p>
</body>
```

Using the Event Object

In modern web browsers the event object includes a `target` property which can be used to retrieve the element which caused event handler execution.[1]

For example:

```
function changeColor(event) {
  var elem = event.target;
  elem.style.backgroundColor = 'red';
}
```

There is also a currentTarget, which is the object currently dealing with the event. To understand the difference, remember that nested DOM objects form a *chain of responsibility*. So clicking on a <p> inside a inside the <body> tag, the p, div, and body all receive the onclick event. The target would be the p element, since that's what the click was actually on. The currentTarget would be either the p, the div, or the body, depending on which element the handler was assigned to.

Using the Document's documentElement Property

The Document's documentElement property gives access to the top of the node tree. This property corresponds to the <html> element on the webpage. You can move from this element down through the elements on the webpage.

For example:

```
document.documentElement.childNodes[0]
```

typically returns the <head> element.

Be very careful when using this property. Different web browsers build slightly different node trees—for example, Firefox has extra Text nodes not contained in the IE node tree.

Checking an Element

If you're not sure you have the correct element, you can test it by checking it's nodeName property and it's nodeValue property. If you've retrieved a node corresponding to an HTML tag, the nodeName should be the HTML tag's name. The value will be null. If you've retrieved a text node the nodeName will be #text and the nodeValue will be the actual text.

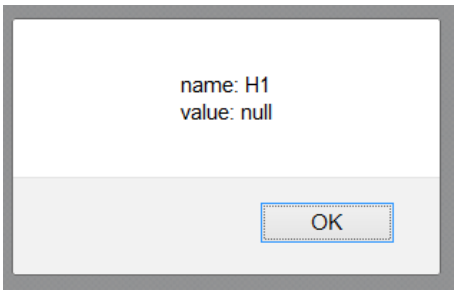
For example running the following function:

```
function check() {
  var heading = document.getElementById("topHeading");
  alert("name: " + heading.nodeName + "\nvalue: " + heading.nodeValue);
}

<body>
<h1 id="topHeading">Id Example</h1>
</body>
```

displays the following alert box:

[1] Pre-IE9 you will need to use a different event object property on Microsoft web browsers, the `srcElement`.



Depending on how you're using this technique you may want to print to the debugger using console.log instead.[2]

Modifying an Element

Once we gain access to an element we can change the element or its children. In this section we look at some simple modifications we can make to an existing element.

Setting an Attribute

The official W3C method for setting an attribute is using the element's setAttribute method. Here is an example of using this to set a picture src and title:

```
function changePhoto() {
    var imageNode = document.getElementById("photo");
    imageNode.setAttribute("src","gates.jpg");
    imageNode.setAttribute("title",
                  "Gates Computer Science Building");
}

<img id="photo" src="memchu.jpg" alt="Photo"
     title="Memorial Church" />
```

As we've already seen, web browsers also support the following simpler syntax:

```
function changePhoto() {
    var imageNode = document.getElementById("photo");
    imageNode.src = "gates.jpg";
    imageNode.title = "Gates Computer Science Building";
}
```

Warning: You'll need to check the documentation to make sure you get the attribute name correct. The DOM attribute names don't always match the HTML attribute names. For example to set the class of an element we need to set the className property as in this function which changes a table data item's class to either "navButtonOn" or "navButtonOff":

[2] On some older web browsers, the console object is only present when the console window is open, if the console window is not open, your code will throw an exception. If you're not careful and leave the code in for production, this can cause unintended errors.

```
function navChange(tdElem,on) {
    if(on)
        tdElem.className='navButtonOn';
   else tdElem.className='navButtonOff';
}
```

Working with Classes

The above technique works with any HTML attribute. If you're specifically working with classes, there is a newer classList property that you can work with. Note as of Summer 2018 caniuse.com reports that this only works in 90% of web browsers.

Each DOM element has a classList property. This property can be used to add, remove, toggle, or replace classes. For example:

```
function toggleColor() {
    var headingNode = document.getElementById("topHeading");
    headingNode.classList.toggle("color");
}

function addUnderline() {
    var headingNode = document.getElementById("topHeading");
    headingNode.classList.add("underdecoration");
}

function removeUnderline() {
    var headingNode = document.getElementById("topHeading");
    headingNode.classList.remove("underdecoration");
}
```

Setting a Style

You can set styles on an element by accessing the style property. This property gives access to a Style object, which can be used to modify an element. Here is function which changes the font-face property of an element on a webpage:

```
function change() {
    var heading = document.getElementById("topHeading");
    heading.style.fontFamily="sans-serif";
}
```

Note that the name of the style property font-family has been converted to JavaScript format fontFamily.[3]

Retrieving a Style

The same technique can be used to retrieve the value of a style. If we're accessing a style we've set explicitly from JavaScript using the technique shown above, we can retrieve the value without any problems.

However, if we're accessing a style set by the HTML and CSS rather than one set explicitly from JavaScript, we may run into problems. If a style has been specifically set using a style attribute-value pair directly on the tag like this:

[3] In other words, all dashes '-' have been removed and all words other than the first word have been capitalized. This capitalization standard is sometimes referred to as "Camel Case".

```
<img src="florence.jpg" id="photo" alt="Photo of Florence"
     style="position: absolute; left: 20px; top: 30px;" />
```

we can retrieve the style value just as if we had set the style from JavaScript—using for example:

```
alert("left is " + document.getElementById("photo").style.left);
```

If instead we set the style using a using the `<style>` tag like this:

```
<style type="text/css">
#photo {position: absolute; left: 20px; top: 30px;}
</style>
…
<img src="florence.jpg" id="photo"  alt="Photo of Florence" />
```

The value we retrieve will be the empty string. What's happening here is that initial style values are set in the DOM only if we assign styles using a style attribute on the tag itself. The rendering engine obviously knows what the style properties are, but they aren't set on the element in the DOM.

Here's the code to retrieve the actual value:

```
document.defaultView.getComputedStyle(
       document.getElementById("photo"),"").left;
```

The key is the `getComputedStyle` function. This function takes two parameters, the first parameter is the object whose property we are trying to access. The second parameter is actually used in case we want the property on a pseudo-element within the object—for example if we're trying to determine the property of the `:first-letter` pseudo-element within the object. This second parameter is almost always set to `""`, signifying that we aren't interested in a specific pseudo-element.

Here is a slightly longer version which is fully W3C compliant and does not take advantage of commonly available simplified syntax—note use of `getPropertyValue` instead of accessing the `left` property directly.

```
document.defaultView.getComputedStyle(
       document.getElementById("photo"),"").getPropertyValue("left");
```

Retrieving a Bounding Rectangle

If you want to determine where a particular item is located, you can get the current position using getClientBoundingRect. The rectangle returned has top, left, width, and height properties. This will give the position of any element relative to the top-left corner of the window. This works whether or not the element is placed using absolute, relative, fixed, or regular static placement.

```
function getRectInfo() {
    var rect = document.getElementById("testPos").getBoundingClientRect();
    alert(`top: ${rect.top}, left: ${rect.left}, height: ${rect.height}, width: ${rect.width}`);
}
```

However, if you are using absolute, relative, or fixed placement, this doesn't necessarily give you the same results as using the getComputedStyle left or top.

If your element is using absolute placement, is not contained within another placed element, and the user has not scrolled, you will get the same results as asking for left and top. If the

use scrolls, however, absolute placement is relative to the top-left corner of the document. the client bounding rect is relative to the top-left corner of the window. When the window is scrolled, these two origins aren't the same anymore. If the absolutely placed element is inside another absolutely, relatively, or fixed placed element, the top and left is relative to the

Changing an Element using innerHTML [Most Web Browsers]

We have several methods available for actually modifying the element. The easiest way to change an element is using the `innerHTML` property. This property was created by Microsoft and is not part of the actual W3C DOM standard. However, it is supported by most web browsers including IE, Firefox, Chrome, and Safari and is considerably easier to use than the W3C standard method.

Here is an example of a function which changes a header to display "*Go* Stanford".

```
function change() {
    var header = document.getElementById("topHeader");
    header.innerHTML="<i>Go</i> <u>Stanford</u>";
}

<body>
<h1 id="topHeader">Example Header</h1>
</body>
```

Warning: IE will not allow change to the `innerHTML` of the following elements:[4] `table`, `thead`, `tfoot`, and `tr`.

You can get around the limitation on the `<table>` tag by surrounding the `<table>` with a `<div>` as follows

```
<div id="example">
<table>
<tr><td>Hello</td></tr>
</table>
</div>
```

and then changing the `<div>` tag's `innerHTML`, rather than the `<table>` tag's `innerHTML`:

```
function test() {
  var divElem = document.getElementById("example");
  divElem.innerHTML = "<table><tr><td>Goodbye</td></tr></table>";
}
```

Changing an Element the W3C DOM Way

Changing an element using the official W3C DOM standard method is much more work. The W3C doesn't provide direct access to an element's contents. Instead we need to create a node and replace the header's original node with our new node.

---

[4] IE11 does seem to support changes on these elements. However, the official documentation still lists the `table`, `thead`, `tfoot`, and `tr` as not changeable. Older versions of IE do not support changes to: `col, colgroup, frameset, html, style, table, tbody, tfoot, thead, title, tr`.

Let's start of by seeing how we can simply replace the header with the unformatted text "Go Stanford":

```
function change() {
    var headerNode = document.getElementById("topHeader");
    var newTextNode = document.createTextNode("Go Stanford");
    var oldTextNode = headerNode.firstChild;
    headerNode.replaceChild(newTextNode,oldTextNode);
}
```

As before, you can see retrieve the header node. However, we can't replace it's contents directly, instead we create a text node using the Document object's createTextNode. Note that we cannot pass HTML tags to createTextNode. Next we retrieve the node we want to replace. This node is the headerNode's child. Finally we call replaceChild on the headerNode passing in the new text node and the node we want to replace.

Now let's see how we can create our formatted "*Go* **Stanford**" text.

```
function change() {
    var headerNode = document.getElementById("topHeader");
    var newHeaderText = document.createElement("span");

    var newNode = document.createElement("i");
    var newText = document.createTextNode("Go");
    newNode.appendChild(newText);
    newHeaderText.appendChild(newNode);

    newText = document.createTextNode(" ");
    newHeaderText.appendChild(newText);

    newNode = document.createElement("b");
    newText = document.createTextNode("Stanford");
    newNode.appendChild(newText);
    newHeaderText.appendChild(newNode);

    var oldTextNode = headerNode.firstChild;
    headerNode.replaceChild(newHeaderText,oldTextNode);
}
```

What a mess!!! Okay, let's walk through this.

- First, I get access to the header node. I also create a element where I can stick the element and text nodes I'm about to create.
- Next I create an <i> node. I have to first create the node itself, then create the text associated with the node. I then append the text into the <i> node and then attach the <i> node to the .
- I create a text node with a single space and append it to the . This is to create the space between "*Go*" and "**Stanford**".
- Next I create a <b> node. I then create the "Stanford" text for the <b> node. I attach the text to the <b> node and then attach the <b> node to my .
- Finally I replace the existing header text with my new .

Reference

Note this reference section summarizes the W3C DOM standard. Not all items listed may be supported in all web browsers.

Document

The Document object supports the following methods for retrieving elements:

`getElementById(`*`elementId`*`)`—This method returns the node corresponding to the HTML element with the given elementId. Notice the "d" in "Id" is not capitalized.

`getElementsByTagName(`*`tagname`*`)`—This method returns an array consisting of all the elements in the document of the given tag type. Notice the name contains "Elements" with an "s". Also notice that the "N" in "TagName" is capitalized.

You may also retrieve the root of the node tree using:

`documentElement`—This property returns the node corresponding to the <html> tag.

The Document object supports the following creation methods:

`createElement(`*`tagName`*`)`—This method creates a new element with the given tag type. For example createElement("i") creates a new <i> tag. Note this new element will have to be explicitly attached to the document after creation.

`createTextNode(`*`data`*`)`—This method creates a new text node which may be attached to the document.

Node

All nodes have the following properties which may be used to get information on the node:

`nodeName`—The name of the node. This will be the tagName if the node corresponds to an HTML element or "#text" for a text node.

`nodeValue`—The value of the node. This will be the null if the node corresponds to an HTML element or the actual text contents a text node.

`nodeType`—The type of the node. Possible values include: 1 = element, 2 = attribute, 3 = text, 4 = cdata, 5 = entity reference, 6 = entity, 7 = processing instruction, 8 = comment, 9 = document node, 10 = document type, 11 = document fragment, 12 = notation. Some nodeType values will only occur in XML documents.

The following properties can be used to move between nodes. These properties are read only.

`parentNode`—The parent node of the current node.

`childNodes`—An array of any child nodes of the current node.

`firstChild`—Accesses the first child node of the current node.

`lastChild`—Accesses the last child node of the current node.

`previousSibling`—Accesses the sibling node before the current node.

`nextSibling`—Accesses the next sibling node which follows the current node.

The following read/only properties can be used to get information about an element:

`attributes`—The attributes of the current node.

`ownerDocument`—The owner Document object of the current node.

The following methods can be used to get information about an element:

`hasChildNodes()`—Used to determine if the node has children.

`hasAttributes()`—Used to determine if the node has attributes.

The following methods can be used to modify an element:

`insertBefore(`*`newChild`*`,`*`refChild`*`)`—Insert a new node before an existing child node.

`replaceChild(`*`newChild`*`,`*`oldChild`*`)`—Replace an existing child node with a new node.

`removeChild(`*`oldChild`*`)`—Remove a child node.

`appendChild(`*`newChild`*`)`—Adds a new child node. The new node is placed last in the child nodes list.

`cloneNode(`*`deepClone`*`)`—Creates a duplicate of the node. Subtrees are recursively cloned if deepClone is true.

Element Node

If a node corresponds to an HTML element it supports a variety of additional methods and properties. The following methods can be used to modify an element's attributes:

`getAttribute(`*`name`*`)`—Retrieves an attribute value.

`setAttribute(`*`name`*`,`*`value`*`)`—Sets an attribute to a particular value.

`removeAttribute(`*`name`*`)`—Removes a given attribute.

The element node also supports versions of the getElementsByTagName, getElementsByClassName, querySelector, and querySelectorAll methods (but not getElementById) found on the Document object.