|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385320) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385320) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385320) |
|  | |
| 9. The Document Object Model  9.1. What is the Document Object Model?  *Copyrighted Material - subject to fair use exception*  The Document Object Model (DOM) is a World Wide Web Consortium standard.  It is defined as a**platform and language independent** i**nterface** that allows programs to **access and update the content, structure and style of documents.**  A web page is a document. This document can be either displayed in the browser window, or as the HTML source. But it is the same document in both cases. The Document Object Model (DOM) provides yet another way to represent, store and manipulate that same document. **The DOM is a fully object-oriented representation of the web page**, and it can be accessed and modified with JavaScript.  The DOM API may be used for both HTML and XML documents.  We’ll focus on HTML here.  9.2. From HTML Document to DOM tree  *Copyright (c) 2014, Rula Khayrallah*  The nested items in an HTML document are represented in the DOM as **a tree of objects**. This **tree**contains **nodes** representing**HTML tags** such as <body> and <p>, and nodes representing **the content inside the tags.**  The nodes that represent the tags are called **element nodes** and the nodes that represent the content inside the tags are called **text nodes**.  And there is also a node representing the whole document.  Let’s look at a simple HTML document, example.html.  Copy this document  and follow along.  Note the special formatting used here with **line breaks inserted inside the HTML tags**.  We’ll use this special  formatting to make it easier to deal with a white space issue in the DOM traversal.  More on that later.  <!DOCTYPE html>  <html          ><head                  ><title>JavaScript for Programmers</title          ></head          ><body                  ><h1>The Document Object Model</h1                  ><h2>How does the document map to the DOM?</h2                  ><p>The nested items in an HTML document are represented in the DOM as a tree of objects.</p          ></body  ></html>    The above is the HTML source of our document (as we can see it in our editor).  We can also open the document in our browser, and this is what we get:                  The corresponding DOM tree for our example.html document is as follows:        Our tree contains three different types of nodes.  At the root of the tree is the green **document node**: it represents the entire document.  The blue nodes are **element nodes** that represent HTML tags.  The red nodes are **text nodes**and they represent the content inside the tags.    9.3. Tree Terminology  *Copyright (c) 2014, Rula Khayrallah*  The DOM tree is just like a family tree:  there are **parents**, **children**, **siblings**, ancestors and descendants.  The only difference is that every node except the root (the document) has **exactly one parent.**  Looking back at our previous example, we can observe the following:  The <html> element node has two children:  the <head> node and the <body> node.  The <head> node is the first child and the <body> node is the last child.  The parent of the <body> node is the <html> node.  The parent of the <head> node is also the <html> node.  The <head> node and the <body> node are siblings.  Similary, the <h1>, <h2> and <p> element nodes are all siblings.  The next sibling of <h2> is <p> and the previous sibling of <h2> is <h1>.  9.4. The DOM in Firebug  *Copyright (c) 2014, Rula Khayrallah*  We can use Firebug to see the various DOM nodes and their relationships.  The following demonstration is illustrated in the screencast below:      We first open the file example.html in Firefox.  Note that if you have a different version of Firefox, the Firefox menu will look different than in the above screencast.  On Windows, you can open example.html or any html document from the File Explorer by right clicking on the file and selecting open with Firefox.  Similarly on a Mac, you can open html documents from the Finder by control clicking on the file and selecting open with Firefox.  We then click on the Firebug icon to open Firebug if it is not already open.  Then we click on the HTML tab as shown below:  If the Firebug side panel is not shown, we can click on the triangle in the upper right corner to show it.  Then we click on the DOM tab in the side panel.  From here on, we can click on any HTML element in the Firebug left panel and the corresponding DOM object will be shown in the right side panel.  Note that the object properties are listed alphabetically.  For example if we highlight the <body> tag in the left panel:  We can see its children in the right panel:  Note that we have childNodes and children.  **childNodes contains all the nodes** (elements and text - blue and red) that are children of the <body> whereas **children contains only the element nodes**(blue only) that are children of <body>.  In this case, since <body> does not have any text node children, childNodes and children have the same elements.  And if we scroll down in the right panel, we can also see its parent and its previous sibling:  Again there are different node and element  properties.  We have parentElement and parentNode and previousElementSibling and previousSibling.  In this case again, they are the same.  Consider however the <html> element.  Its**parent node is the document node**(green) and it has**no parent element**(null).  9.5. White space in the DOM  *Copyrighted Material - subject to fair use exception*  In our example, we used a special formatting in our HTML source file to deal with a white space issue in the DOM.  The different browsers deal with the white space (or new lines) inside the HTML source in an inconsistent manner.  That can make **the node traversal of the DOM tree unpredictable**.  In Firefox, all white space in the document is represented as extra text nodes in the DOM.  That includes the new line character between <html> and <head> below.  <html>        <head>  So to avoid ending up with extra text nodes we put our line breaks inside the tags, where white space is ignored.  We'll see later how to use a framework like jQuery to deal with these inconsistencies between browsers.  Another option is to use the **element traversal**interface where the text nodes are ignored.  We'll see examples of both node and element traversals later in this module.  We will no longer use the special formatting of HTML source files.  We'll just keep in mind that the DOM tree may have some extra text nodes containing white space.  9.6. Layout Engines and the DOM  *Copyrighted Material - subject to fair use exception*  The different browsers rely on layout engines to parse HTML into a DOM.  In general, layout engines are responsible for parsing the source HTML, generating the DOM representation and displaying the content.  The different layout engines implement the DOM standards to **varying degrees of compliance.**  Webkit is the layout engine currently used by Safari (both desktop and mobile) and the Android browser.  Blink is currently used by Chrome and Opera, Gecko is used by Firefox and Trident is used by Internet Explorer.  To figure out what DOM features a given browser supports, we need to go back to the underlying layout engine documentation.  In this course, we’ll cover the features that are supported by most engines.  9.7. Selecting Elements: where to start?  *Copyrighted Material - subject to fair use exception*  When we want to access objects in an HTML page, we start with the document object, window.document or simply document.  The document object has two properties: documentElement and body.  document.documentElement gives us access to the top most <html> element.  **document.body gives us access to the <body> element.**  Let’s go back and open our example.html file in Firefox, then go the Firebug console, and type the following.  Note that you only need to type what comes after the >>> prompt.  The text in blue is the output you get.  >>> document  Document *example.html*  >>> document.documentElement  <html>  >>> document.body  <body>  9.8. Selection by id  *Copyrighted Material - subject to fair use exception*  There are several methods defined on the document object to access individual elements.  The recommended method is the simplest one: **getElementById().**  When we need to access a specific element, we give that element an HTML id attribute, and then we can look it up in your program using that id.  Note that any HTML element can have an id attribute. The value of this attribute must be **unique** within the document — no two elements in the same document can have the same id.  Let’s modify our first example file to add some ids.   We'll save the new document in the file idexample.html.  <!DOCTYPE html>  <html>      <head>          <title>JavaScript for Programmers</title>      </head>      <body>          <h1 **id = "main"**>The Document Object Model</h1>          <h2 **id = "subtitle"**>How does the document map to the DOM?</h2>          <p **id = "par1**"> The nested items in an HTML document are represented in the DOM as a tree of objects.</p>      </body>  </html>  Let’s open idexample.html in Firefox, then go the console, and type the following.  Here again you only need to type what comes after the  >>> prompt.  The text in blue is the output you get.  >>> document.getElementById('main');  <h1 id="main">  >>> document.getElementById('subtitle');  <h2 id="subtitle">  >>> document.getElementById('par1');  <p id="par1">  If the id does not exist in the document, getElementById returns null:  >>> document.getElementById('noid');  null  Note that the id is case sensitive.  If we specify “Main” instead of “main”, we get null:  >>> document.getElementById('Main');  null    9.9. Selection by type  *Copyrighted Material - subject to fair use exception*  We can select all elements of a specified type (or tag name) using the **getElementsByTagName()**method.  This method is **defined on the document object as well as on  individual elements.**  It returns **an array like object** containing all elements with a given tag, starting at the document level or at the specified element level:  To get all the elements with a <p> tag in the document, we write:  >>> document.getElementsByTagName('p');  HTMLCollection[p#par1]  To get all the elements with an <h1> tag in the document, we write:  >>> document.getElementsByTagName('h1');  HTMLCollection[h1#main]  We can use \* as a wild card to get ALL tags in a document:  >>> document.getElementsByTagName('\*');  HTMLCollection[html, head, title, body, h1#main, h2#subtitle, p#par1]  Note that **we get an array like object containing all the matching elements**.  To access one particular element, we can index it with square brackets [ ].  Indexing starts at 0.  >>> document.getElementsByTagName('\*')[0];  <html>  Our search does not have to start at the document level.  We can start at a given element and look for all the nested elements that satisfy the tag search:  >>> var myMain = document.getElementById("main");  >>> myMain  <h1 id="main">   The variable myMain now points to the element object <h1>.  We can look for any tags starting at myMain.  Of course there is none.  >>> myMain.getElementsByTagName('\*');  HTMLCollection[]  Let’s examine an element with nested tags such as the <body> element.  We'll first use the body property of the document  to access the <body> element object and then we'll invoke the getElementsByTagName method on that body object:  var  bodyObj = document.body;  >>> **bodyObj**.getElementsByTagName('\*');  HTMLCollection[h1#main, h2#subtitle, p#par1]  Or, writing the above two statements in one line:  >>> document.body.getElementsByTagName('\*')  HTMLCollection[h1#main, h2#subtitle, p#par1]  9.10. Selection by class name  *Copyrighted Material - subject to fair use exception*  Similarly, we can access elements by class name.  The class attribute specifies one or more class names for an element.  A class name usually refers to a style class defined in a style sheet and used to apply the same presentation style to different elements.  We'll see examples of style classes when we introduce style sheets.  In HTML5, the class attribute can be used on any HTML element.  We can select all elements of a specified class using the **getElementsByClassName()** method.  This method is  **defined on the document object as well as on individual elements.**  It **returns an array like object** containing all matching elements, starting at the document level or at the specified element level.  **getElementsByClassName()** takes one argument, but that argument may specify **multiple class names separated by a space**. Only elements that **include all of the specified class names in their class attribute are matched.** The order of the identifiers does not matter.  Let’s look at the following html source document classexample.html:  <!DOCTYPE html>  <html>          <head>                  <title>JavaScript for Programmers</title>          </head>          <body>                  <h1 class = "important"> Layout Engines</h1>                  <p id = "mainidea" >                          <span id = "first" class = "info">The different browsers rely on layout engines. </span>                          <span id = "second" class = "important info">                                  The different layout engines implement the DOM standards to varying degrees of compliance.                          </span>                  </p>          </body>  </html>  You can follow along by opening the document in Firefox and typing the commands below at the Firebug console.  To get all the elements with the 'important' class, we write:  >>>document.getElementsByClassName('important');  HTMLCollection[h1.important, span#second.important]  We get the h1 heading element as well as the second span element.  Note that **we get an array like object containing all the matching elements**.  To access one particular element, we can index it  with square brackets [ ].  Indexing starts at 0.  >>>document.getElementsByClassName('important')[0];  <h1 class="important">  To get all the elements with the 'info' class:  >>> document.getElementsByClassName('info');  HTMLCollection[span#first.info, span#second.important]  We get the first span element (with a class = ‘info’) as well as the second span element with class = ‘important info’.  To get all elements with both the 'info' and 'important' classes:  >>> document.getElementsByClassName('info important');  HTMLCollection[span#second.important]  We get the second span element with class = ‘important info’.  Note that the order of the class names specified did not matter.  The method is also defined on element nodes (not just on document). To get all the elements that have a class of ‘important’ inside the ‘main idea’ paragraph, we can write the following:  >>> var mainPar = document.getElementById('mainidea');  >>> **mainPar**.getElementsByClassName('important');  HTMLCollection[span#second.important]  We can chain the two method invocations above into one statement, replacing mainPar by its value:  >>> **document.getElementById('mainidea')**.getElementsByClassName('important');  HTMLCollection[span#second.important]  Note that the h1 heading element does not appear here even though it has a class of 'important'.  The search for the 'important' class was restricted to the descendants of the paragraph element with id = 'mainidea'.  9.11. Node Based Traversal  *Copyrighted Material - subject to fair use exception*  Once we have selected an element from a Document, we sometimes need to find related items. There are two approaches to do that.  One is based on a node traversal and the other is based on an element traversal.  The node traversal interface was the original one.  It provides the following node object properties:   * parentNode * childNodes * firstChild * lastChild * nextSibling * previousSibling   Let’s illustrate these properties with our previous idexample.html.  Note that we did not use the special formatting to get rid of the line breaks between tags here.  idexample.html.  <!DOCTYPE html>  <html>          <head>                  <title>JavaScript for Programmers</title>          </head>          <body>                  <h1 id = "main">The Document Object Model</h1>                  <h2 id = "subtitle">How does the document map to the DOM?</h2>                  <p id = "par1"> The nested items in an HTML document are represented in the DOM as a tree of objects.                  </p>          </body>  </html>  Again you can follow along by opening the document in Firefox and typing the commands below at the Firebug console.  >>> document.body.parentNode;  <html>  >>> document.parentNode;  null  The document node has no parent node so the value of parentNode is null.  >>> document.body.childNodes;  NodeList[<TextNode textContent="\n ">, h1#main, <TextNode textContent="\n ">, h2#subtitle, <TextNode textContent="\n ">, p#par1, <TextNode textContent="\n \n">]  document.body.childNodes is an array like object that represents all the node children of <body>.  Here it contains 3 element nodes <h1>, <h2> and <p> as well as 4 extra text nodes representing the line breaks in the source document.  We can access each child node by its index.  Indexing starts at 0.  >>> document.body.childNodes[0];  <TextNode textContent="\n ">  >>> document.body.firstChild;  <TextNode textContent="\n ">  document.body.firstChild is the same as document.body.childNodes[0].  >>> document.body.lastChild;  <TextNode textContent="\n \n">  >>> document.body.childNodes[1];  <h1 id="main">  And to get the children nodes of <h1>:  >>> document.body.childNodes[1].childNodes;  NodeList[<TextNode textContent="The Document Object Model">]  >>> document.body.childNodes[3];  <h2 id="subtitle">  >>> document.body.childNodes[3].nextSibling;  <TextNode textContent="\n ">  >>> document.body.childNodes[3].childNodes;  NodeList[<TextNode textContent="How does the document map to the DOM?">]  >>> document.body.childNodes[3].previousSibling;  <TextNode textContent="\n ">    ????  As you can see, navigating the DOM from node to node is tricky because of the extra newline text nodes.  9.12. Element Based Traversal  *Copyrighted Material - subject to fair use exception*  The more convenient alternative is to use the newer element traversal interface that implements an element only based navigation and ignores text nodes.  We can use the following element properties for the element traversal:  firstElementChild  lastElementChild  previousElementSibling  nextElementSibling  childElementCount  In addition to the above standard properties, most current browsers implement the children property which returns only the children element nodes.  Again let’s see how these properties work through an example.  >>> document.body.children;  HTMLCollection[h1#main, h2#subtitle, p#par1]  document.body.children is an array like object that represents the element children of <body>.  Here it contains 3 element nodes <h1>, <h2> and <p>.  We can access each child element by its index.  Indexing starts at 0.  >>> document.body.children[0];  <h1 id="main">  >>> document.body.children[1];  <h2 id="subtitle">  >>> document.body.children[2];  <p id="par1">  >>> document.body.children[3];  undefined  >>> document.body.childElementCount;  3  >>> document.body.firstElementChild;  <h1 id="main">  You can see that document.body.firstElementChild is the same as document.body.children[0].  >>> document.body.lastElementChild;  <p id="par1">  >>> document.body.nextElementSibling;  null  The <body> element has no next sibling so the value is null.  >>> document.body.previousElementSibling;  <head >  Note however that the text nodes do not appear in this navigation.  The <h1> element has no (element) children.  >>> document.body.firstElementChild;  <h1 id="main">  >>> document.body.firstElementChild.children;  HTMLCollection[]  9.13. HTML Attributes and Element Properties  *Copyrighted Material - subject to fair use exception*  HTML elements consist of a tag name and a set of name-value pairs known as attributes. We have seen the id and the class attributes in our previous examples. We have also used the src attribute of the <script> tag.  HTML attributes are not case sensitive, but JavaScript property names are. To convert an attribute name to the JavaScript property, we need to convert it to lowercase,  The attribute name class is a reserved word in JavaScript.  So the corresponding property name is className.  Let’s create a new HTML source file that contains an image.  The <img> tag has a src attribute that is similar to the src attribute we’ve encountered in <script> .  It points to the file that contains the image to be displayed.  To follow along, you can download the frown.gif and smile.gif files from Resources and save them in the same location as your HTML source file.  Or you can create your own images.  imageexample.html  <!DOCTYPE html>  <html>    <head>        <meta charset="utf-8">        <title>JavaScript for Programmers</title>    </head>    <body>        <h2>HTML Attributes and Element Properties</h2>        <p> The attribute values of the HTML elements are available as properties .</p>        <p> We'll change the src attribute of the image below to go from a frown to a smile. </p>        <img id="myimage" src="frown.gif">    </body>  </html>  Open the file imageexample.html in Firefox.  You’ll get the following:  Then enter the following at the Firebug console prompt:  >>> var myMood = document.getElementById("myimage");  We first get the element corresponding to the <img> tag.  We select it by id, which is the simplest way.  We then query its src attribute.  >>> myMood.src  "file:///C:/Users/Rula/Documents/CS22A/html/frown.gif"  Now we change its src attribute to point to the smiley image:  >>> myMood.src = "smile.gif";  And the image is changed in the browser as shown below:    9.14. Element Content  *Copyrighted Material - subject to fair use exception*  We have seen how to get to the element object with methods such as getElementsById and getElementsByClassName.  We have also seen how to change an element attribute.  But how do we access and possibly modify what’s in these elements?  We'll take a look at two properties of element objects that allow us do that:  innerHTML and textContent.  **innerHTML:**  The innerHTML property returns the content of an element as a string of markup.  Basically it **returns everything in between the opening and closing tags of that element, including tags of nested elements, if any.**  Let’s go back to our idexample.html source file, open it in Firefox, and type the following at the console prompt:  >>> var myMain = document.getElementById('main');  >>> myMain  <h1 id="main">  The variable myMain now points to the element object <h1>.  The innerHTML property will return everything between the opening and closing <h1> tags:  >>> myMain.innerHTML  "The Document Object Model"  >>> var myParagraph = document.getElementById('par1');  >>> myParagraph  <p id="par1">  The variable myParagraph now points to the element object <p>.  >>> myParagraph.innerHTML  " The nested items in an HTML document are represented in the DOM as a tree of objects."  Note that so far, the innerHTML property has returned text, with no HTML markup.  That’s just because the elements we were accessing did not have any nested tags.  Now let’s look at the innerHTML property of the body element:  >>> document.body.innerHTML  "                                 <h1 id="main">The Document Object Model</h1>                                 <h2 id="subtitle">How does the document map to the DOM?</h2>                                 <p id="par1"> The nested items in an HTML document are represented in the DOM as a tree of objects.</p>  "  You can see that it includes everything between <body> and </body> including the white space and the nested tags and their text.  **The innerHTML property is writable, so we can use it to modify the element**.  >>> var myParagraph = document.getElementById('par1');  >>> myParagraph  <p id="par1">  >>> myParagraph.innerHTML =  'We just changed the whole paragraph.';  And the text corresponding to the paragraph on the web page is modified:  We can even delete the entire content of the document by setting the innerHTML of the <body> element to the empty string.  >>> document.body.innerHTML = '';  And the whole document disappears.  There are several issues we need to be aware of when using innerHTML:  The first issue is that if we are inserting plain text, we need to be aware that it will be **parsed as HTML.** So we’ll need to use &lt; instead of <  and &gt; instead of > and &amp; instead of & and so on.  The second and most important issue is that using the innerHTML property comes with a security risk.  Consider the following code where we are assigning the innerHTML of some element to some variable that is outside your control.  **someVariable could be any HTML code that invokes any script.**  document.getElementById(‘someId’).innerHTML = someVariable;  For that reason, it is recommended not to use innerHTML when inserting plain text.  **textContent:**  The textContent property gives us access to the **text content of the specified element,** **and all its descendants**.  Let's go back to our idexample.  Note that you can reload the original page by pressing F5 (or refresh).  First let's take a look at the paragraph text content:  >>> var myParagraph = document.getElementById('par1');  >>> myParagraph.textContent  "The nested items in an HTML document are represented in the DOM as a tree of objects."  **The textContent property is also writable, so we can use it to modify the element text**.  >>> myParagraph.textContent =   'We can change the paragraph with textContent.';  And the web page is instantly modified to reflect the change in the paragraph.    What happens when an element has descendants?  Let's take a look at the <body> element to illustrate the textContent property in that case.  Again, you may want to reload the original page before you proceed with the example.  >>> document.body.textContent  "                                 The Document Object Model                                 How does the document map to the DOM?                                  The nested items in an HTML document are represented in the DOM as a tree of objects.    "  textContent concatenates the individual text contents of <body> and all its descendants.  Let's see what happens when we assign a new value to the  textContent property here:  >>> document.body.textContent = 'And this is the new text!';  The web page is instantly modified to reflect the change.  The whole text has been replaced.  Let's click on the HTML tab in the Firebug console to see what happened to the <h1> <h2> and <p> elements that were descendants of document.body:  We can see that the <h1> <h2> and <p> elements are all gone.  9.15. User Input  *Copyright (c) 2014, Rula Khayrallah*  We have seen how to access and modify the text in our document with the innerHTML, and textContent properties.  Often we need to access text that the user enters and these properties do not help us.  We can use the **input tag** in HTML source documents to define a user entry field.  This tag may or may not be inside a <form> tag.  Let’s look at the following example:  calculator.html  <!DOCTYPE html>  <html>          <head>                  <meta charset="utf-8">                  <title>My Simple Calculator</title>          </head>          <body>                  <h2>Let's add some numbers! </h2>                  <p>Please enter two numbers:  </p>  **<input id = "first" type="number">**                  <br>  **<input id = "second" type="number">**                  <p> And the answer is:</p>                  <p id="answer"></p>          </body>  </html>  We have defined two <input> elements "first" and "second" where the user may enter numbers.  We would like to compute the sum of these numbers and display it in the "answer" element.  To follow along, copy and save the source document calculator.html on your system then open it in Firefox.  The following is displayed:  Let’s enter two numbers, say 6 and 18.  How do we access these numbers in our program? The properties innerHTML and textContent don’t give us what we need.  They just return an empty string.  >>> document.getElementById('first').innerHTML  ""  >>> document.getElementById('first').textContent  ""  **The value property may be used to access the input that the user enters.**  >>> document.getElementById('first').**value**  "6"  >>> document.getElementById('second').**value**  "18"  Note first that the value property is giving us a **string** not a number.  **To convert it to a number, we can use the conversion function Number()**.  We'll  to convert the two strings to numbers and save them in two variables.  >>> var firstNumber = Number(document.getElementById('first').value);  >>> var secondNumber = Number(document.getElementById('second').value);  Then we compute the sum:  >>> var myAnswer = firstNumber + secondNumber;  >>> myAnswer  24  And to show myAnswer on the web page, we write it in the 'answer' element using textContent.  >>> document.getElementById('answer').textContent = myAnswer;  The web page is updated accordingly:  Next week we’ll see how to add event handlers so that the answer is automatically updated whenever the input changes. |  |