**JavaScript HTML DOM Navigation**

With the HTML DOM, you can navigate the node tree using node relationships.

**DOM Nodes**

According to the W3C HTML DOM standard, everything in an HTML document is a node:

* The entire document is a document node
* Every HTML element is an element node
* The text inside HTML elements are text nodes
* Every HTML attribute is an attribute node
* All comments are comment nodes



With the HTML DOM, all nodes in the node tree can be accessed by JavaScript.

New nodes can be created, and all nodes can be modified or deleted.

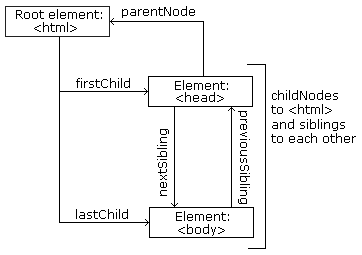
**Node Relationships**

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

* In a node tree, the top node is called the root (or root node)
* Every node has exactly one parent, except the root (which has no parent)
* A node can have a number of children
* Siblings (brothers or sisters) are nodes with the same parent

<html>  
  
  <head>  
      <title>DOM Tutorial</title>  
  </head>  
  
  <body>  
      <h1>DOM Lesson one</h1>  
      <p>Hello world!</p>  
  </body>  
  
</html>



From the HTML above you can read:

* <html> is the root node
* <html> has no parents
* <html> is the parent of <head> and <body>
* <head> is the first child of <html>
* <body> is the last child of <html>

and:

* <head> has one child: <title>
* <title> has one child (a text node): "DOM Tutorial"
* <body> has two children: <h1> and <p>
* <h1> has one child: "DOM Lesson one"
* <p> has one child: "Hello world!"
* <h1> and <p> are siblings

**Navigating Between Nodes**

You can use the following node properties to navigate between nodes with JavaScript:

* parentNode
* childNodes[*nodenumber*]
* firstChild
* lastChild
* nextSibling
* previousSibling

**Warning !**

A common error in DOM processing is to expect an element node to contain text.

In this example: **<title>DOM Tutorial</title>**, the element node <title> does not contain text. It contains a **text node** with the value "DOM Tutorial".

The value of the text node can be accessed by the node's **innerHTML** property, or the **nodeValue**.

**Child Nodes and Node Values**

In addition to the innerHTML property, you can also use the childNodes and nodeValue properties to get the content of an element.

The following example collects the node value of an <h1> element and copies it into a <p> element:

**Example**

<html>  
<body>  
  
<h1 id="intro">My First Page</h1>  
  
<p id="demo">Hello!</p>  
  
<script>  
var myText = document.getElementById("intro").childNodes[0].nodeValue;  
document.getElementById("demo").innerHTML = myText;  
</script>  
  
</body>  
</html>

In the example above, getElementById is a method, while childNodes and nodeValue are properties.

In this tutorial we use the innerHTML property. However, learning the method above is useful for understanding the tree structure and the navigation of the DOM.

Using the firstChild property is the same as using childNodes[0]:

**Example**

<html>  
<body>  
  
<h1 id="intro">My First Page</h1>  
  
<p id="demo">Hello World!</p>  
  
<script>  
myText = document.getElementById("intro").firstChild.nodeValue;  
document.getElementById("demo").innerHTML = myText;  
</script>  
  
</body>  
</html>

**DOM Root Nodes**

There are two special properties that allow access to the full document:

* document.body - The body of the document
* document.documentElement - The full document

**Example**

<html>  
<body>  
  
<p>Hello World!</p>  
<div>  
<p>The DOM is very useful!</p>  
<p>This example demonstrates the <b>document.body</b> property.</p>  
</div>  
  
<script>  
alert(document.body.innerHTML);  
</script>  
  
</body>  
</html>

**Example**

<html>  
<body>  
  
<p>Hello World!</p>  
<div>  
<p>The DOM is very useful!</p>  
<p>This example demonstrates the <b>document.documentElement</b> property.</p>  
</div>  
  
<script>  
alert(document.documentElement.innerHTML);  
</script>  
  
</body>  
</html>

**The nodeName Property**

The nodeName property specifies the name of a node.

* nodeName is read-only
* nodeName of an element node is the same as the tag name
* nodeName of an attribute node is the attribute name
* nodeName of a text node is always #text
* nodeName of the document node is always #document

**Note:** nodeName always contains the uppercase tag name of an HTML element.

**The nodeValue Property**

The nodeValue property specifies the value of a node.

* nodeValue for element nodes is undefined
* nodeValue for text nodes is the text itself
* nodeValue for attribute nodes is the attribute value

**The nodeType Property**

The nodeType property returns the type of node. nodeType is read only.

The most important node types are:

|  |  |
| --- | --- |
| **Element type** | **NodeType** |
| Element | 1 |
| Attribute | 2 |
| Text | 3 |
| Comment | 8 |
| Document | 9 |

**JavaScript HTML DOM Node List**

A node list is a collection of nodes

**HTML DOM Node List**

The getElementsByTagName() method returns a **node list**. A node list is an array-like collection of nodes.

The following code selects all <p> nodes in a document:

**Example**

var x = document.getElementsByTagName("p");

The nodes can be accessed by an index number. To access the second <p> node you can write:

y = x[1];

**Note:** The index starts at 0.

**HTML DOM Node List Length**

The length property defines the number of nodes in a node list:

**Example**

var myNodelist = document.getElementsByTagName("p");  
document.getElementById("demo").innerHTML = myNodelist.length;

Example explained:

1. Get all <p> elements in a node list
2. Display the length of the node list

The length property is useful when you want to loop through the nodes in a node list:

**Example**

Change the background color of all <p> elements in a node list:

var myNodelist = document.getElementsByTagName("p");  
var i;  
for (i = 0; i < myNodelist.length; i++) {  
    myNodelist[i].style.backgroundColor = "red";  
}

**A node list is not an array!**  
A node list may look like an array, but it is not. You can loop through the node list and refer to its nodes like an array. However, you cannot use Array Methods, like valueOf() or join() on the node list.

**JavaScript HTML DOM EventListener**

**The addEventListener() method**

**Example**

Add an event listener that fires when a user clicks a button:

document.getElementById("myBtn").addEventListener("click", displayDate);

The addEventListener() method attaches an event handler to the specified element.

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers to one element.

You can add many event handlers of the same type to one element, i.e two "click" events.

You can add event listeners to any DOM object not only HTML elements. i.e the window object.

The addEventListener() method makes it easier to control how the event reacts to bubbling.

When using the addEventListener() method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

You can easily remove an event listener by using the removeEventListener() method.

**Syntax**

*element*.addEventListener(*event, function, useCapture*);

The first parameter is the type of the event (like "click" or "mousedown").

The second parameter is the function we want to call when the event occurs.

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

**Add an Event Handler to an Element**

**Example**

Alert "Hello World!" when the user clicks on an element:

*element*.addEventListener("click", function(){ alert("Hello World!"); });

You can also refer to an external "named" function:

**Example**

Alert "Hello World!" when the user clicks on an element:

*element*.addEventListener("click", myFunction);  
  
function myFunction() {  
    alert ("Hello World!");  
}

**Add Many Event Handlers to the Same Element**

The addEventListener() method allows you to add many events to the same element, without overwriting existing events:

**Example**

*element*.addEventListener("click", myFunction);  
*element*.addEventListener("click", mySecondFunction);

You can add events of different types to the same element:

**Example**

*element*.addEventListener("mouseover", myFunction);  
*element*.addEventListener("click", mySecondFunction);  
*element*.addEventListener("mouseout", myThirdFunction);

**Add an Event Handler to the Window Object**

The addEventListener() method allows you to add event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that support events, like the xmlHttpRequest object.

**Example**

Add an event listener that fires when a user resizes the window:

window.addEventListener("resize", function(){  
    document.getElementById("demo").innerHTML = *sometext*;  
});

**Passing Parameters**

When passing parameter values, use an "anonymous function" that calls the specified function with the parameters:

**Example**

*element*.addEventListener("click", function(){ myFunction(p1, p2); });

**Event Bubbling or Event Capturing?**

There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In *bubbling* the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the <div> element's click event.

In *capturing* the outer most element's event is handled first and then the inner: the <div> element's click event will be handled first, then the <p> element's click event.

With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter:

addEventListener(*event*, *function*, *useCapture*);

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

**Example**

document.getElementById("myP").addEventListener("click", myFunction, true);  
document.getElementById("myDiv").addEventListener("click", myFunction, true);

**The removeEventListener() method**

The removeEventListener() method removes event handlers that have been attached with the addEventListener() method:

**Example**

*element*.removeEventListener("mousemove", myFunction);

**Browser Support**

The numbers in the table specifies the first browser version that fully supports these methods.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Method** |  |  |  |  |  |
| addEventListener() | 1.0 | 9.0 | 1.0 | 1.0 | 7.0 |
| removeEventListener() | 1.0 | 9.0 | 1.0 | 1.0 | 7.0 |

**Note:** The addEventListener() and removeEventListener() methods are not supported in IE 8 and earlier versions and Opera 6.0 and earlier versions. However, for these specific browser versions, you can use the attachEvent() method to attach an event handlers to the element, and the detachEvent() method to remove it:

*element.*attachEvent*(event, function);  
element.*detachEvent*(event, function);*

**Example**

Cross-browser solution:

var x = document.getElementById("myBtn");  
if (x.addEventListener) {                    // For all major browsers, except IE 8 and earlier  
    x.addEventListener("click", myFunction);  
} else if (x.attachEvent) {                  // For IE 8 and earlier versions  
    x.attachEvent("onclick", myFunction);  
}

**HTML DOM Event Object Reference**

**JavaScript Closures**

JavaScript variables can belong to the **local** or **global** scope.

Private variables can be made possible with **closures**.

**Global Variables**

A function can access all variables defined **inside** the function, like this:

**Example**

function myFunction() {  
    var a = 4;  
    return a \* a;  
}

But a function can also access variables defined **outside** the function, like this:

**Example**

var a = 4;  
function myFunction() {  
    return a \* a;  
}

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the window object.

Global variables can be used (and changed) by all scripts in the page (and in the window).

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variables created **without** the keyword **var**, are always global, even if they are created inside a function.

**Variable Lifetime**

Global variables live as long as your application (your window / your web page) lives.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

**A Counter Dilemma**

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

**Example**

var counter = 0;  
  
function add() {  
    counter += 1;  
}  
  
add();  
add();  
add();  
  
// the counter is now equal to 3

The counter should only be changed by the add() function.

The problem is, that any script on the page can change the counter, without calling add().

If I declare the counter inside the function, nobody will be able to change it without calling add():

**Example**

function add() {  
    var counter = 0;  
    counter += 1;  
}  
  
add();  
add();  
add();  
  
// the counter should now be 3, but it does not work !

It did not work! Every time I call the add() function, the counter is set to 1.

**A JavaScript inner function can solve this.**

**JavaScript Nested Functions**

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function **plus()** has access to the **counter** variable in the parent function:

**Example**

function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();      
    return counter;   
}

This could have solved the counter dilemma, if we could reach the **plus()** function from the outside.

We also need to find a way to execute **counter = 0** only once.

**We need a closure.**

**JavaScript Closures**

Remember self-invoking functions? What does this function do?

**Example**

var add = (function () {  
    var counter = 0;  
    return function () {return counter += 1;}  
})();  
  
add();  
add();  
add();  
  
// the counter is now 3

**Example Explained**

The variable **add** is assigned the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.