Tabla de contenido

What is Javascript?

JavaScript (*JS for short*) is the programming language that enables web pages to respond to user interaction beyond the basic level. It was created in 1995, and is today one of the most famous and used programming languages.

Basics

Statement

A statement is more casually (and commonly) known as a *line of code*. That's because statements tend to be written on individual lines. As such, programs are read from top to bottom, left to right. You might be wondering what code (also called source code) is. That happens to be a broad term which can refer to the whole of the program or the smallest part. Therefore, a line of code is simply a line of your program.

Here is a simple example:

```
var hello = "Hello";
var world = "World";

// Message equals "Hello World"
var message = hello + " " + world;
```

This code can be executed by another program called an *interpreter* that will read the code, and execute all the statements in the right order

Comments

Comments are statements that will not be executed by the interpreter, comments are used to mark annotations for other programmers or small descriptions of what your code does, thus making it easier for others to understand what your code does.

In Javascript, comments can be written in 2 different ways:

Line starting with `//`:

```
// This is a comment, it will be ignored by the interpreter
var a = "this is a variable defined in a statement";
```

Section of code starting with `/*`and ending with `*/`, this method is used for multi-line comments:

```
/*
This is a multi-line comment,
it will be ignored by the interpreter
*/
var a = "this is a variable defined in a statement";
```

Variables

In programming, variables are containers for values that change. Variables can hold all kind of values and also the results of computations. Variables have a name and a value separated by an equals sign (=). Variable names can be any letter or word, but bear in mind that there are restrictions from language to language of what you can use, as some words are reserved for other functionality.

Let's check out how it works in Javascript, The following code defines two variables, computes the result of adding the two and defines this result as a value of a third variable.

```
var x = 5;
var y = 6;
var result = x + y;
```

Types

JavaScript is a *"loosely typed"* language, which means that you don't have to explicitly declare what type of data the variables are. You just need to use the `var` keyword to indicate that you are declaring a variable, and the interpreter will work out what data type you are using from the context, and use of quotes.

Equality

For example, assume:

```
var foo = 42;
var bar = 42;
var baz = "42";
var qux = "life";
```

`foo == bar` will evaluate to `true` and `baz == qux` will evaluate to `false`, as one would expect. However, `foo == baz` will *also* evaluate
to `true` despite `foo` and `baz` being different types. Behind the scenes
the `==` equality operator attempts to force its operands to the same type before determining their equality. This is in contrast to the `===` equality operator.
The `===` equality operator determines that two variables are equal if they are of the same type *and* have the same value. With the same assumptions as before, this means that `foo === bar` will still evaluate to `true`, but `foo === baz` will now evaluate to `false`. `baz === qux` will still evaluate to `false`.

Numbers

Operators

You can apply mathematic operations to numbers using some basic operators like:

- **Addition**: `c = a + b`
- **Subtraction**: `c = a - b`
- **Multiplication**: `c = a * b`
- **Division**: `c = a / b`

You can use parentheses just like in math to separate and group expressions: `c = (a / b) + d`

Advanced Operators

Some advanced operators can be used, such as:

- **Modulus (division remainder)**: `x = y % 2`
- **Increment**: Given a = 5
  - `c = a++`, Results: c = 5 and a = 6
  - `c = ++a`, Results: c = 6 and a = 6
- **Decrement**: Given a = 5
  - `c = a--`, Results: c = 5 and a = 4
  - `c = --a`, Results: c = 4 and a = 4

Conditional

The easiest condition is an if statement and its syntax is `if(condition){ do this … }`. The condition has to be true for the code inside the curly braces to be executed. You can for example test a string and set the value of another string dependent on its value:

```
var country = 'France';
var weather;
var food;
var currency;

if(country === 'England') {
    weather = 'horrible';
    food = 'filling';
    currency = 'pound sterling';
}

if(country === 'France') {
    weather = 'nice';
    food = 'stunning, but hardly ever vegetarian';
    currency = 'funny, small and colourful';
}

if(country === 'Germany') {
    weather = 'average';
    food = 'wurst thing ever';
    currency = 'funny, small and colourful';
}

var message = 'this is ' + country + ', the weather is ' +
            weather + ', the food is ' + food + ' and the ' +
            'currency is ' + currency;
```

Loops

For

The easiest form of a loop is the for statement. This one has a syntax that is similar to an if statement, but with more options:

```
for(condition; end condition; change){
    // do it, do it now
}
```

Lets for example see how to execute the same code ten-times using a `for` loop:

```
for(var i = 0; i < 10; i = i + 1){
    // do this code ten-times
}
```

While

While Loops repetitively execute a block of code as long as a specified condition is true.

```
while(condition){
    // do it as long as condition is true
}
```

For example, the loop in this example will repetitively execute its block of code as long as the variable i is less than 5:

```
var i = 0, x = "";
while (i < 5) {
    x = x + "The number is " + i;
    i++;
}
```

Do…while

The do...while statement creates a loop that executes a specified statement until the test condition evaluates to be false. The condition is evaluated after executing the statement. Syntax for do… while is

```
do{
    // statement
}
while(expression) ;
```

Lets for example see how to print numbers less than 10 using `do...while` loop:

```
var i = 0;
do {
    document.write(i + " ");
    i++; // incrementing i by 1
} while (i < 10);
```

Arrays

Arrays are a fundamental part of programming. An array is a list of data. We can store a lot of data in one variable, which makes our code more readable and easier to understand. It also makes it much easier to perform functions on related data.

The data in arrays are called **elements**.

Here is a simple array:

```
// 1, 1, 2, 3, 5, and 8 are the elements in this array
var numbers = [1, 1, 2, 3, 5, 8];
```

So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. An **index** refers to a spot in the array. indices logically progress one by one, but it should be noted that the first index in an array is 0, as it is in most languages. Brackets [] are used to signify you are referring to an index of an array.

```
// This is an array of strings
var fruits = ["apple", "banana", "pineapple", "strawberry"];

// We set the variable banana to the value of the second element of
// the fruits array. Remember that indices start at 0, so 1 is the
// second element. Result: banana = "banana"
var banana = fruits[1];
```

Arrays have a property called length, and it's pretty much exactly as it sounds, it's the length of the array.

```
var array = [1 , 2, 3];

// Result: l = 3
var l = array.length;
```

Functions

Functions, like variables, must be declared. Let's declare a function `double` that accepts an **argument** called `x` and **returns** the double of x :

```
function double(x) {
    return 2 * x;
}
```

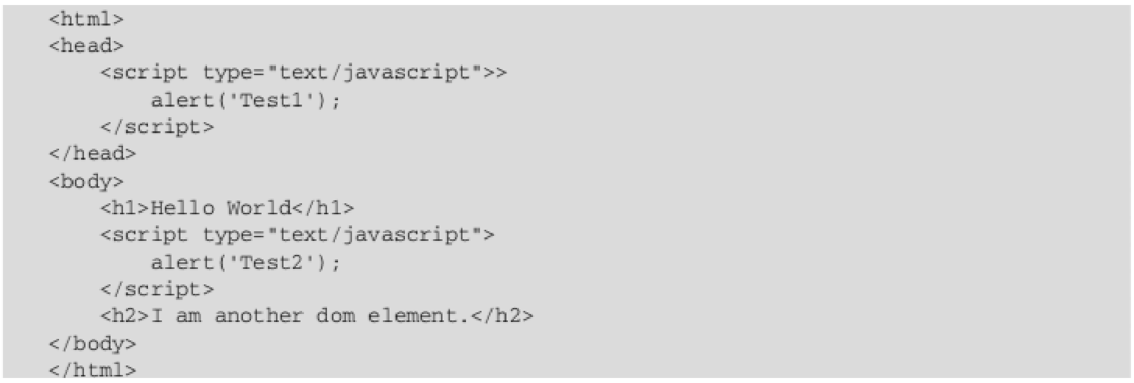
Note: the function above **may** be referenced before it has been defined.

Functions are also values in JavaScript; they can be stored in variables (just like numbers, strings, etc ...) and given to other functions as arguments :

```
var double = function(x) {
    return 2 * x;
};
```

Inline Scripts

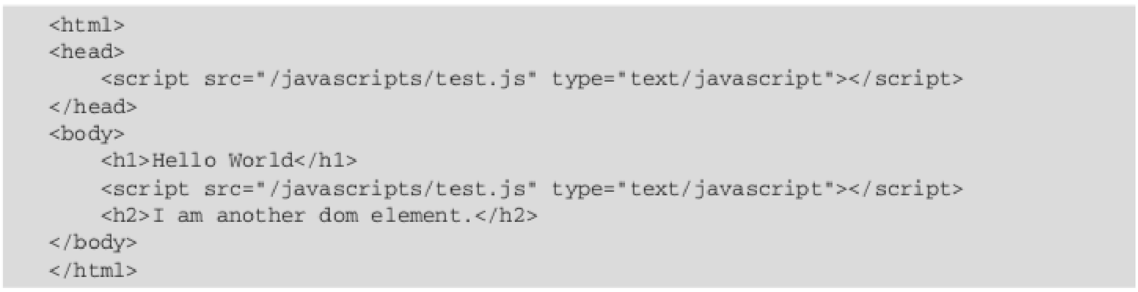
When I refer to inline scripts, I mean scripts meant to be executed *as soon as they are encountered* if the page were read from top to bottom. These can appear in both the <head> and <body> areas of the page. Here's an example:



```
<html>
<head>
    <script type="text/javascript">>
        alert('Test1');
    </script>
</head>
<body>
    <h1>Hello World</h1>
    <script type="text/javascript">
        alert('Test2');
    </script>
    <h2>I am another dom element.</h2>
</body>
</html>
```

External Scripts

Earlier I showed that scripts can be placed directly in the contents of the HTML document, and they can also be included from external files as in the following example. Externally referenced scripts execute at exactly the same points as inline scripts. The following example would behave the same as the previous example, despite that the script is located in another document.



```
<html>
<head>
    <script src="/javascripts/test.js" type="text/javascript"></script>
</head>
<body>
    <h1>Hello World</h1>
    <script src="/javascripts/test.js" type="text/javascript"></script>
    <h2>I am another dom element.</h2>
</body>
</html>
```

The only difference here is that in practice your page may take slightly longer to load because the browser has to pause execution and reach out to the network or file system to grab your external file. For big scripts over slow connections this can be noticeable delay. For small scripts as in this one here, it really doesn't make much difference.

Event-Driver Scripts

The browser has a rich event model that you can use, which includes events for when the DOM itself is loaded, and also for when the entire page including all image and script assets have been downloaded. As you progress as a developer, you will come to rely on this event model.

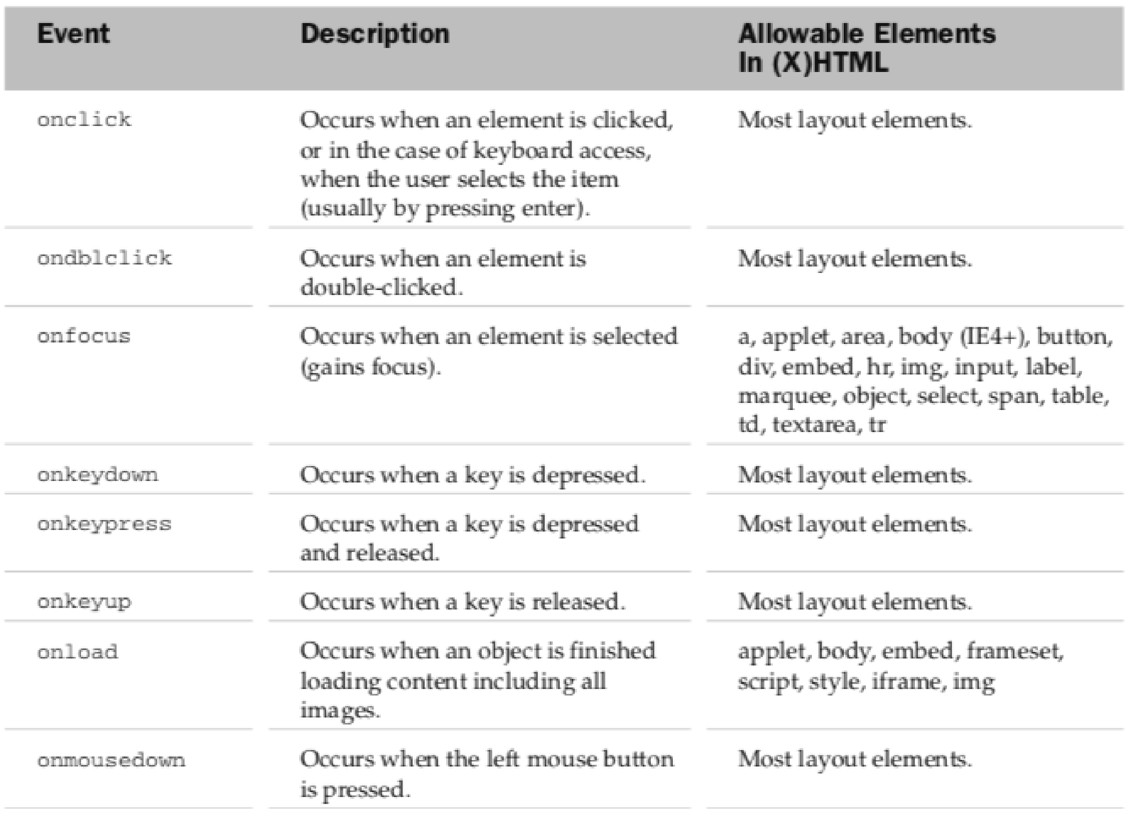
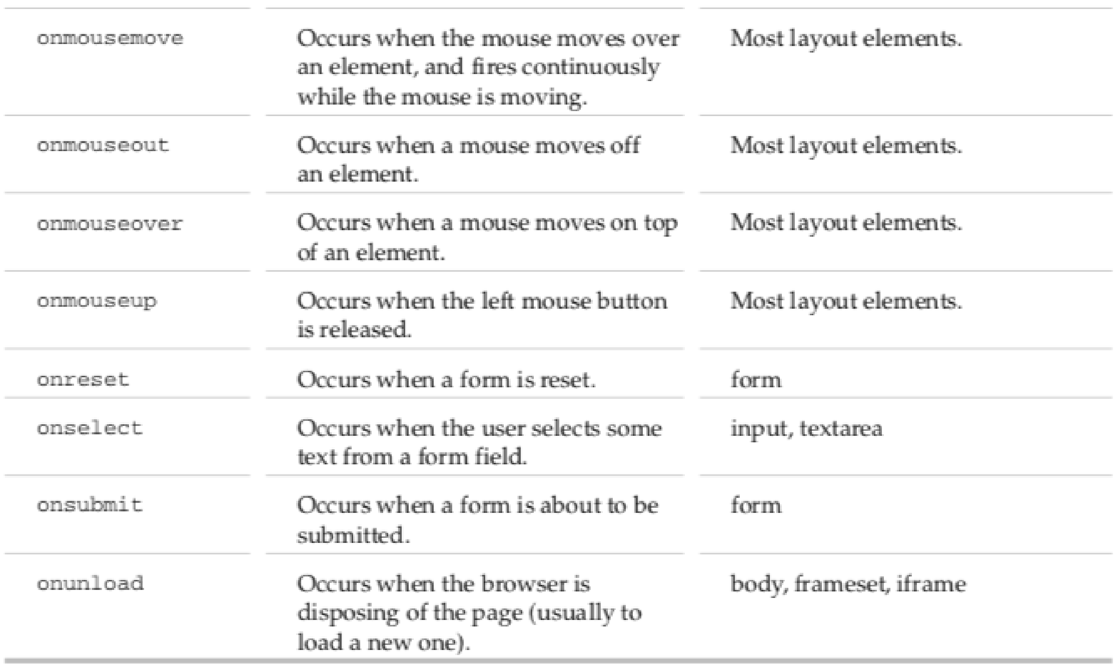
The simplest way to have your script execute after the page has loaded is to harness the onload event of the page. There are two ways to do this. The simplest way is to use the HTML onload attribute of the <body> element. The following example uses concepts I haven't discussed yet, such as *functions*. Feel free to read over this and return to it later after reading Chapter 5.

```
<!-- Using the onload event on the body tag to control script execution -->
<html>
<head>
<script type="text/javascript">
function myFunction() {
    alert('Hello!');
}
</script>
</head>
<body onload="myFunction()">
    <h1>Hello World!</h1>
</body>
</html>
```

Another way you might trigger JavaScript to execute is in response to user actions. For example, you might want some script to execute when a user clicks a button or mouses over a heading. Like the onload event, you can attach these events by using the event attribute in HTML:

```
<!-- Using the onclick event of a button to trigger script execution -->
<html>
<head>
<script type="text/javascript">

function myFunction() {
    alert('Hello!');
}

</script>
</head>
<body>
   <button onclick="myFunction()">Click me!</button>
</body>
</html>
```

There are some rules as to which events will fire on which elements. For example, there is no blur event for non-visible elements on a page, and the change event is limited to a small subset of elements belonging to forms. The table that follows provides a short list of common event bindings and which elements they are typically supported on in HTML and XHTML:

Event	Description	Allowable Elements In (X)HTML
onclick	Occurs when an element is clicked, or in the case of keyboard access, when the user selects the item (usually by pressing enter).	Most layout elements.
ondblclick	Occurs when an element is double-clicked.	Most layout elements.
onfocus	Occurs when an element is selected (gains focus).	a, applet, area, body (IE4+), button, div, embed, hr, img, input, label, marquee, object, select, span, table, td, textarea, tr
onkeydown	Occurs when a key is depressed.	Most layout elements.
onkeypress	Occurs when a key is depressed and released.	Most layout elements.
onkeyup	Occurs when a key is released.	Most layout elements.
onload	Occurs when an object is finished loading content including all images.	applet, body, embed, frameset, script, style, iframe, img
onmousedown	Occurs when the left mouse button is pressed.	Most layout elements.
onmousemove	Occurs when the mouse moves over an element, and fires continuously while the mouse is moving.	Most layout elements.
onmouseout	Occurs when a mouse moves off an element.	Most layout elements.
onmouseover	Occurs when a mouse moves on top of an element.	Most layout elements.
onmouseup	Occurs when the left mouse button is released.	Most layout elements.
onreset	Occurs when a form is reset.	form
onselect	Occurs when the user selects some text from a form field.	input, textarea
onsubmit	Occurs when a form is about to be submitted.	form
onunload	Occurs when the browser is disposing of the page (usually to load a new one).	body, frameset, iframe

Javascript in URL’s

If you wanted to execute JavaScript in the URL of a hyperlink, you could do so using this technique like so:

```
<a href=”javascript:alert(‘Hello World!’)”>Hello World</a>
```

That being said, this is a holdover from the very first generation of browsers and there are more elegant ways to achieve the result that I’ll cover in Chapter 12.

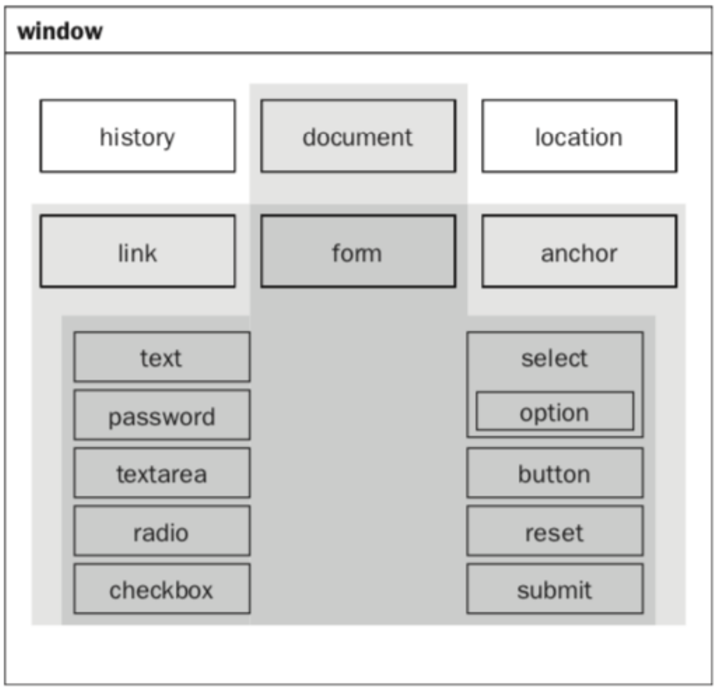
DOM Model

The DOM serves as an object representation of all the elements and information pertaining to the layout of the page. Technically speaking, a browser does not *need* a DOM to render a web page, but it is required if it wants to allow JavaScript to read or write information to the page. Historically this has been the most inconsistently implemented feature of web browsers, but in recent years this problem has been mitigated thanks in large part to the work that the World Wide Web Consortium (W3C) has done in documenting a standard for DOMs (http://www.w3.org/DOM/).

In JavaScript, your DOM can be accessed simply by referencing the global object document. To access the body element, you can typically just reference document.body. If you wanted to access the HTML content of the <body> element as a string, you could access the innerHTML property of that element (document.body. innerHTML). This is the power of the DOM. If you think of your page as a hierarchical object model, it becomes something you can represent easily in a JavaScript object.

The Legacy Object Model

This design provided a hierarchical set of nodes, beginning with the window object and, moving inward, included a document object, a forms collection, links collection, and anchors collection. Inside the forms collection was a number of node types representing the form controls (text box, radio button, and so on)



Basic Model Plus Images

Shortly after Microsoft introduced IE 3, Netscape followed up with Navigator 3. The most important addition to the DOM at this time was the images collection and the power to script <img> tag src properties

With Navigator 4, Netscape again rewrote the rules for browser scripting. They invented a new event model (which included the first semblance of an event object), along with the concept of event *bubbling*, which was not adopted by Microsoft. They also introduced new capabilities at the window level, allowing developers to move and resize the window at will. It was now possible to make a browser window full screen, entirely from script.

Despite slow beginnings, some credit has to go Microsoft for the important innovations introduced by Internet Explorer 4, released a short time after Navigator 4. This was one of those leap-frog events like the introduction of Ajax, which brought browser scripting to a new level. For starters, they began representing the *entire* document in the object model. All HTML nodes could be accessed via the DOM, and elements could be styled with CSS from script. Microsoft introduced several new collections, including document.all, which was an array containing every element on the page. These were accessible via their IDs, and it made every element a *scriptable* object. If you had a form field with an ID of myForm, you could access it via the all object like this:

```
document.all.myForm
```

In addition to a styleSheets collection, IE 4 allowed developers to set style properties directly onto objects. It also let developers modify the *contents* of the DOM via four new object properties: innerHTML, outerHTML, innerText, and outerText. This single piece of functionality opened up worlds of possibility to developers wanting to design truly *dynamic* HTML documents. Developers could now rewrite portions of the DOM simply by typing:

```
document.all.myTableId.outerHTML = "<p>Now I'm a paragraph instead of a table!</p>";
```

The Document Tree

Indeed, the best way to think of a document *is* like a tree. Every node on the page is contained inside another node, moving all the way up to the root of the document, the document object (documentElement according to the W3C). You can navigate up and down this structure using the built-in APIs and calculate things like screen position by taking advantage of this hierarchical relationship.

When a browser encounters this page, it will begin the DOM by creating a document element that will contain two nodes: html and body, as can be seen in the preceding tag structure. Inside each of these, it will populate a hierarchical set of nodes based on their parent-child relationships in HTML. For example, under body, the ul element (representing *unordered list*), will belong to body, and four lis will belong to the ul node. There will be no direct relationship between body and li, but you can get to li by first identifying ul.

If you try to represent the preceding structure in its DOM object format, with each node appearing on its own line, the result might look like this:

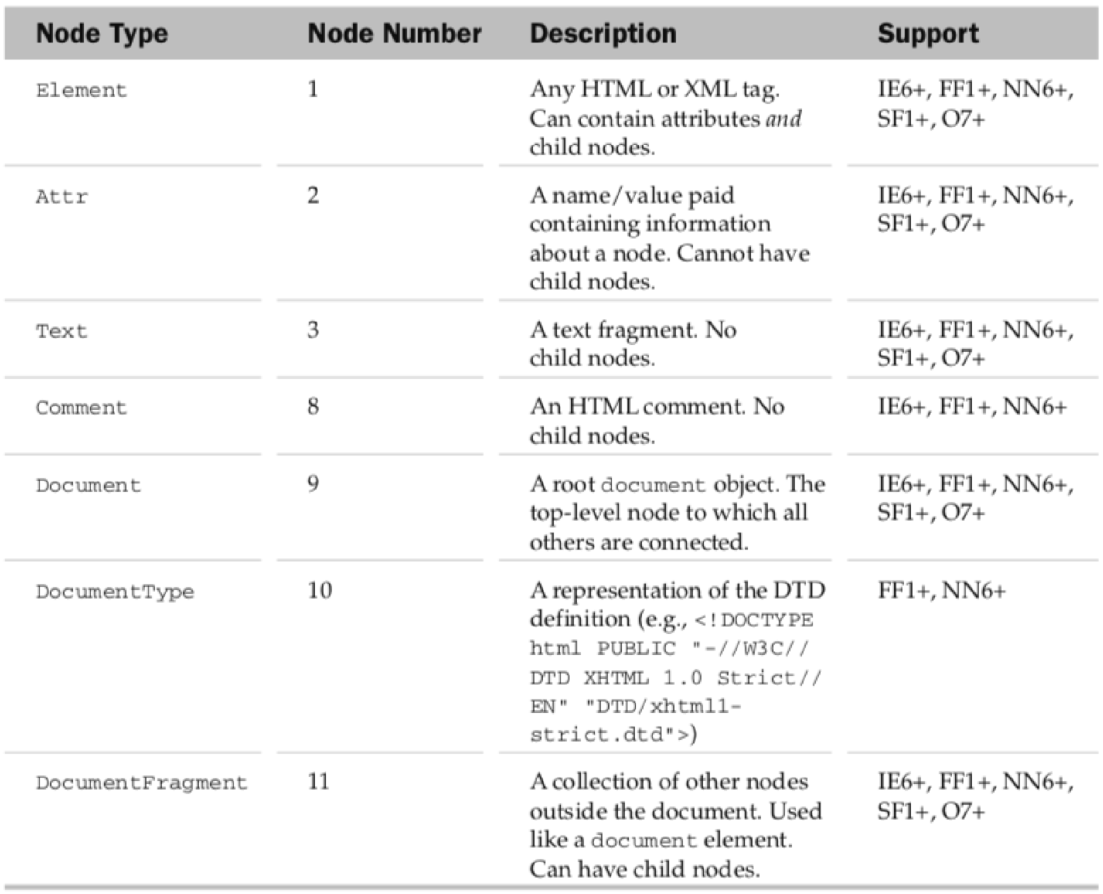
```
document

+--html

  +--head

    +--title

      +--"My List"

  +--body

    +--h1 class="myHeading"

      +--"My List!"

      +--ul id="myUnorderedList"
```

```
+--li
  +--”Item 1”
+--li
  +--”Item 2”
+--li
  +--”Item 3”
+--li
  +--”Item 4”
```

The hierarchical nature of the document is seen in the relationships between parent and child nodes. This is reflected in the APIs provided by the W3C and Microsoft, in that it’s easy, once you *have* a reference to a node, to move up and down this chain using those relationships.

Node Types

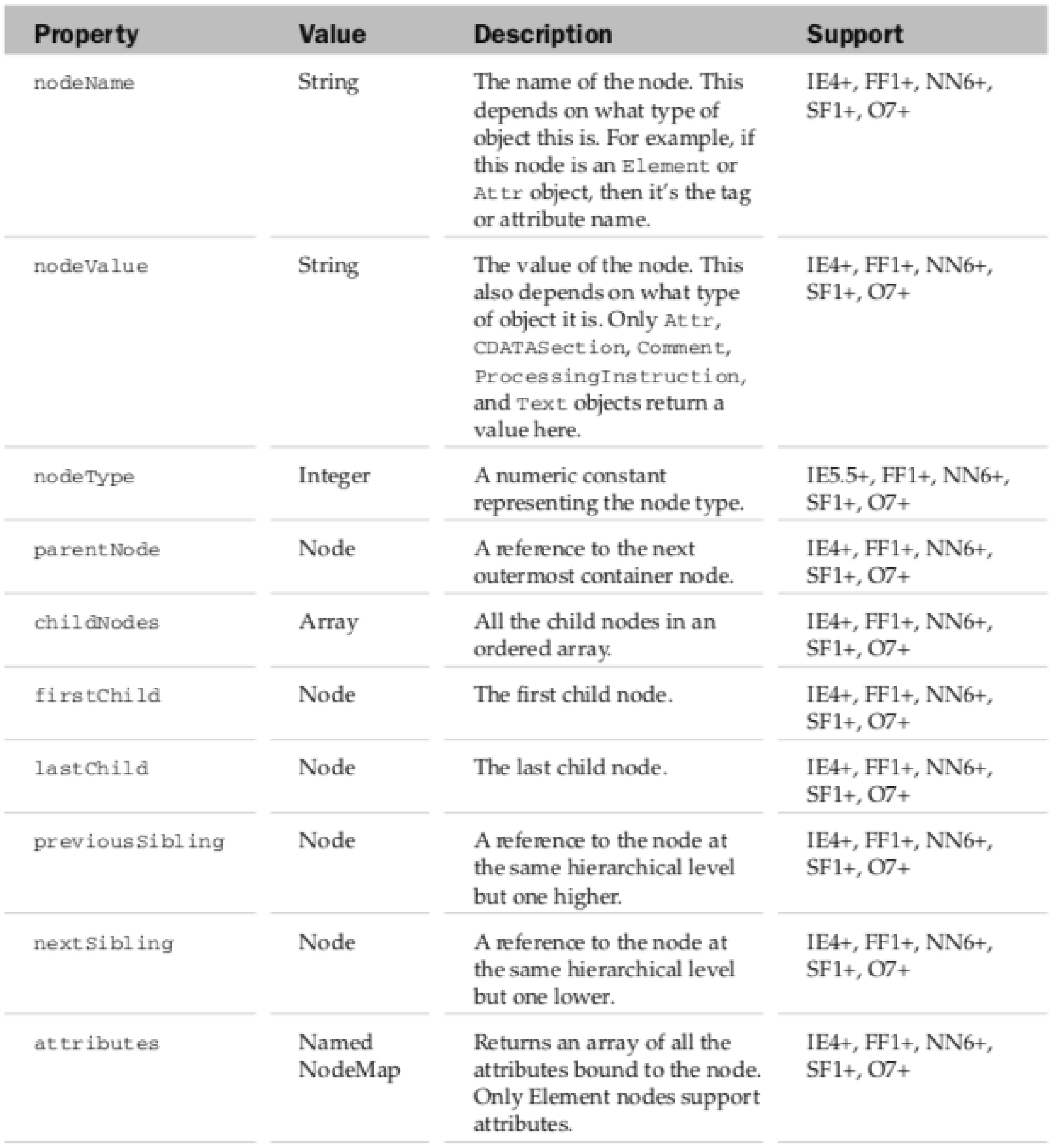
According to the W3C specification Level 2, there are 12 types of nodes in an HTML or XML document. Only seven of these relate specifically to HTML, however. Really only three of these are used with any frequency: document, element, and text.



Node Type	Node Number	Description	Support
`Element`	1	Any HTML or XML tag. Can contain attributes *and* child nodes.	IE6+, FF1+, NN6+, SF1+, O7+
`Attr`	2	A name/value paid containing information about a node. Cannot have child nodes.	IE6+, FF1+, NN6+, SF1+, O7+
`Text`	3	A text fragment. No child nodes.	IE6+, FF1+, NN6+, SF1+, O7+
`Comment`	8	An HTML comment. No child nodes.	IE6+, FF1+, NN6+
`Document`	9	A root `document` object. The top-level node to which all others are connected.	IE6+, FF1+, NN6+, SF1+, O7+
`DocumentType`	10	A representation of the DTD definition (e.g., `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "DTD/xhtml1-strict.dtd">`)	FF1+, NN6+
`DocumentFragment`	11	A collection of other nodes outside the document. Used like a `document` element. Can have child nodes.	IE6+, FF1+, NN6+, SF1+, O7+

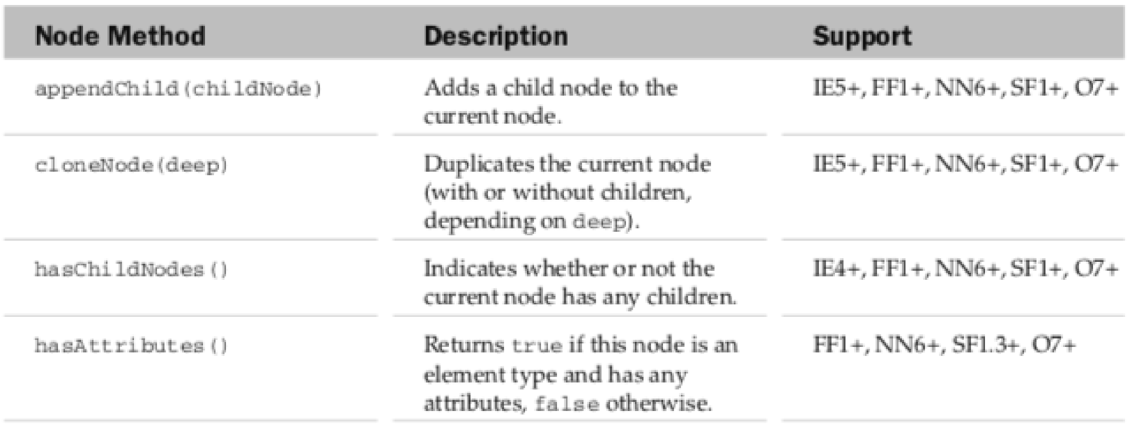
Node Properties

Nodes have many properties as well. Ones that are not methods either contain specific details about the node or references to other nodes nearby in the tree:

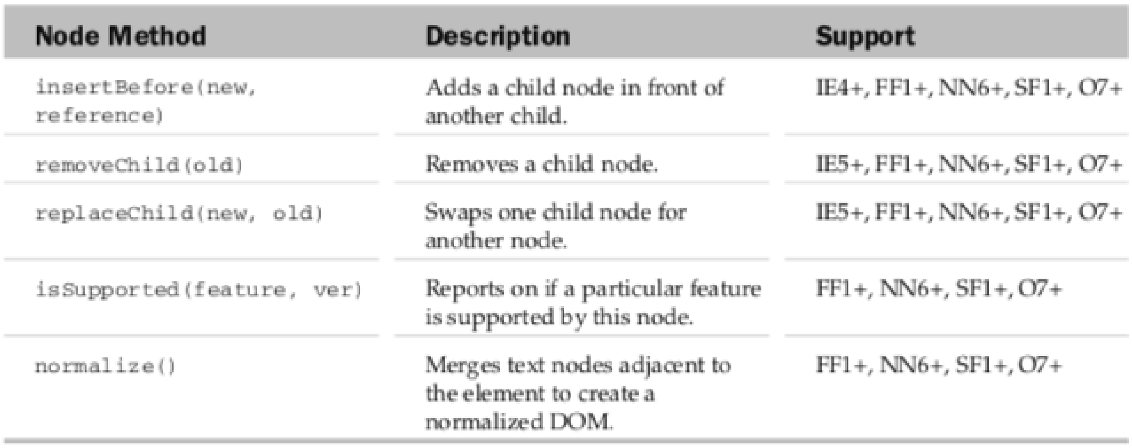


Property	Value	Description	Support
`nodeName`	String	The name of the node. This depends on what type of object this is. For example, if this node is an `Element` or `Attr` object, then it's the tag or attribute name.	IE4+, FF1+, NN6+, SF1+, O7+
`nodeValue`	String	The value of the node. This also depends on what type of object it is. Only `Attr`, `CDATASection`, `Comment`, `ProcessingInstruction`, and `Text` objects return a value here.	IE4+, FF1+, NN6+, SF1+, O7+
`nodeType`	Integer	A numeric constant representing the node type.	IE5.5+, FF1+, NN6+, SF1+, O7+
`parentNode`	Node	A reference to the next outermost container node.	IE4+, FF1+, NN6+, SF1+, O7+
`childNodes`	Array	All the child nodes in an ordered array.	IE4+, FF1+, NN6+, SF1+, O7+
`firstChild`	Node	The first child node.	IE4+, FF1+, NN6+, SF1+, O7+
`lastChild`	Node	The last child node.	IE4+, FF1+, NN6+, SF1+, O7+
`previousSibling`	Node	A reference to the node at the same hierarchical level but one higher.	IE4+, FF1+, NN6+, SF1+, O7+
`nextSibling`	Node	A reference to the node at the same hierarchical level but one lower.	IE4+, FF1+, NN6+, SF1+, O7+
`attributes`	Named NodeMap	Returns an array of all the attributes bound to the node. Only Element nodes support attributes.	IE4+, FF1+, NN6+, SF1+, O7+

Node Methods



Node Method	Description	Support
`appendChild(childNode)`	Adds a child node to the current node.	IE5+, FF1+, NN6+, SF1+, O7+
`cloneNode(deep)`	Duplicates the current node (with or without children, depending on `deep`).	IE5+, FF1+, NN6+, SF1+, O7+
`hasChildNodes()`	Indicates whether or not the current node has any children.	IE4+, FF1+, NN6+, SF1+, O7+
`hasAttributes()`	Returns `true` if this node is an element type and has any attributes, `false` otherwise.	FF1+, NN6+, SF1.3+, O7+



Node Method	Description	Support
`insertBefore(new, reference)`	Adds a child node in front of another child.	IE4+, FF1+, NN6+, SF1+, O7+
`removeChild(old)`	Removes a child node.	IE5+, FF1+, NN6+, SF1+, O7+
`replaceChild(new, old)`	Swaps one child node for another node.	IE5+, FF1+, NN6+, SF1+, O7+
`isSupported(feature, ver)`	Reports on if a particular feature is supported by this node.	FF1+, NN6+, SF1+, O7+
`normalize()`	Merges text nodes adjacent to the element to create a normalized DOM.	FF1+, NN6+, SF1+, O7+

Traversing DOM

As you know by now, the DOM is a relational structure with the properties of each node pointing to others nearby it in the tree. At the top of this tree you have the document element, an instance of HTMLDocument. To get a reference to the <HTML> property of a web page, you refer to the document .documentElement attribute. You can also get a reference to this node from any element by using the ownerDocument.documentElement property:

```
// this will be true if myElement resides in the same document as window.document

myElement.ownerDocument == document
myElement.ownerDocument.documentElement == document.documentElement
```

Let's use a simple HTML document to serve as an example before looking at document traversal:

```
<html> <head>

<title>My List</title> </head>

<body>
<h1 class="myHeading">My List!</h1> <ul id="myUnorderedList">

<li>Item 1</li> <li>Item 2</li> <li>Item 3</li> <li>Item 4</li>

</ul> </body>

</html>
```

Once you have a reference to the <html> node, you have to begin using DOM node properties to access anything below that. There are a few ways to get a reference to nodes below the current one. You've already seen these properties: firstChild, lastChild, and childNodes[]. Since the HTML node only *has* two elements (<head> and <body>), firstChild and lastChild will retrieve these nicely:

var head = document.documentElement.firstChild; var body = document.documentElement.lastChild;

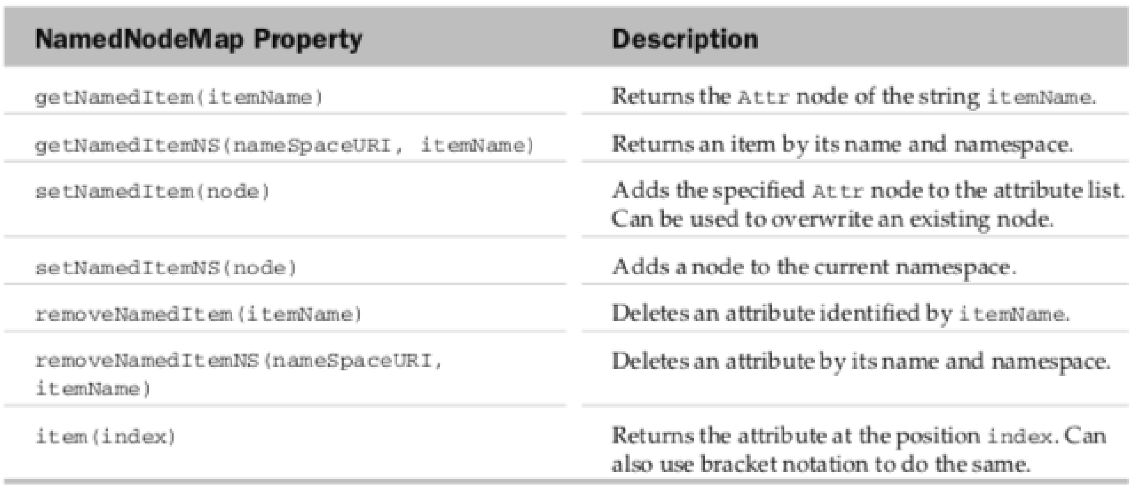
Element Attributes

Once you have a reference to a DOM node, you can look at it in detail to see what attributes it contains. There are a few ways to get this information.

First of all, what do I mean when I say *attribute*? For the most part I'm referring to HTML attributes — but because elements can have expando properties, there can be many more attributes on a DOM node than just the HTML-defined attributes. However, let's start small. Consider the following HTML node:

<img src="logo.gif" class="logoImg" onclick="alert('hi');" >

If you had a reference to this node, you could access the class attribute via the attributes object on the node that returns a NamedNodeMap of Attr elements. In addition to behaving like a NodeMap with bracket notation, this collection has the following members:



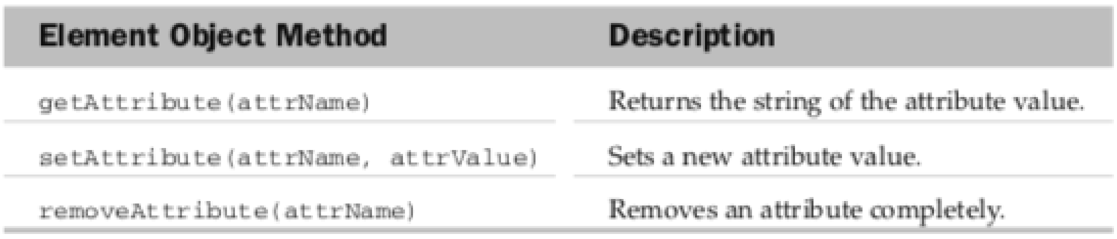
NamedNodeMap Property	Description
getNamedItem(itemName)	Returns the Attr node of the string itemName.
getNamedItemNS(nameSpaceURI, itemName)	Returns an item by its name and namespace.
setNamedItem(node)	Adds the specified Attr node to the attribute list. Can be used to overwrite an existing node.
setNamedItemNS(node)	Adds a node to the current namespace.
removeNamedItem(itemName)	Deletes an attribute identified by itemName.
removeNamedItemNS(nameSpaceURI, itemName)	Deletes an attribute by its name and namespace.
item(index)	Returns the attribute at the position index. Can also use bracket notation to do the same.

You can query a specific attribute by using the getNamedItem(itemName) method like this. This will return an Attr attribute object, *not* the value of the attribute. To get the value, use the property nodeValue:

// Will return "logoImg" myImg.attributes.getNamedItem("class").nodeValue

The nodeValue property is also writable. To change its value just set it to something else:
myImg.attributes.getNamedItem("class").nodeValue = "newImgClass";

Another, if problematic way of doing this is to use the attribute *helpers* on the element itself:



Element Object Method	Description
`getAttribute(attrName)`	Returns the string of the attribute value.
`setAttribute(attrName, attrValue)`	Sets a new attribute value.
`removeAttribute(attrName)`	Removes an attribute completely.

Finding Specific Elements

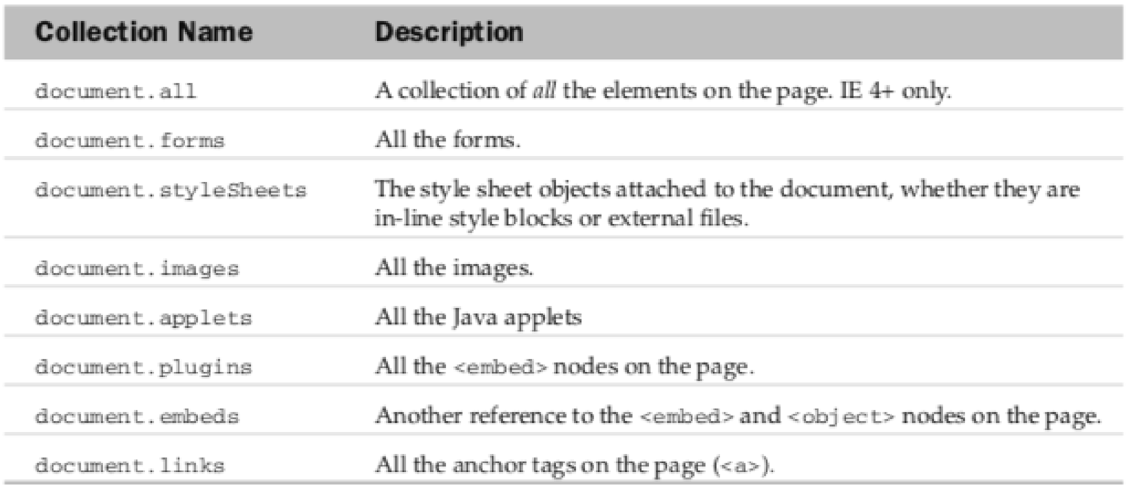
Now that you understand the over structure of the DOM, you'll look at how to target specific pieces of it. Both DOM Level 0 and later versions provide ways to "query" the document for specific nodes. Once you have a reference to a node, you can use it to change the look and feel of the document or change its content. There are four general ways to target specific elements in the DOM. The first is using the DOM Level 0 element collections, and the other ways involve the methods getElementsByName(), getElementsByTagName(), and the popular getElementById().

Element Collectios

In the very first versions of the DOM, there was only one way to target a specific element, and that was to use one of several array-like objects that grouped together elements of a specific type. The earliest of these was the document.forms[] array, which contained references to all of the forms in the document. You could iterate over the forms collection by using the array length property and passing the index to the array:

for (var fIndex = 0; fIndex < document.forms.length; fIndex++) var theFormObj = document.forms[fIndex];

You can also reference the form by using its name attribute as the argument: var myFormObj = document.forms["theFormName"];

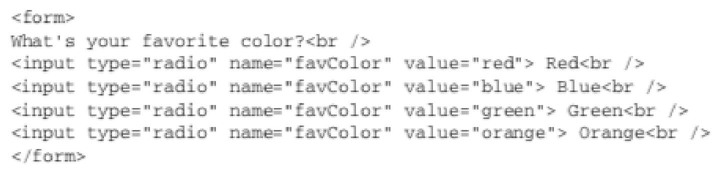


Collection Name	Description
`document.all`	A collection of *all* the elements on the page. IE 4+ only.
`document.forms`	All the forms.
`document.styleSheets`	The style sheet objects attached to the document, whether they are in-line style blocks or external files.
`document.images`	All the images.
`document.applets`	All the Java applets
`document.plugins`	All the `<embed>` nodes on the page.
`document.embeds`	Another reference to the `<embed>` and `<object>` nodes on the page.
`document.links`	All the anchor tags on the page (`<a>`).

getElementsByName

The document.getElementsByName(name) static function returns an array of elements that have a name attribute that matches the argument. This is useful in particular for form radio buttons because multiple elements grouped together can have the same name attribute. If no elements are found with the specified name, an array of zero-length is returned.

For example, say you have a radio group with the name "favColor":



```
<form>
What's your favorite color?<br />
<input type="radio" name="favColor" value="red"> Red<br />
<input type="radio" name="favColor" value="blue"> Blue<br />
<input type="radio" name="favColor" value="green"> Green<br />
<input type="radio" name="favColor" value="orange"> Orange<br />
</form>
```

You can get a reference to the group by using its name:

```
var favColorGroup = document.getElementsByName("favColor");
```

Iterating over this list is now just like iterating over an array:

```
for (var cI = 0; cI < favColorGroup.length; cI++)

  document.write("Color: " + favColorGroup[cI].value + "<br />");
```

Also, because it's an array, you can access each item via bracket notation:

```
   favColorGroup[1].value        // "blue"
```

This is very similar to your next DOM query function: getElementsByTagName().

getElementsByTagName

The second utility function you need to know about for querying the DOM is getElementsByTagName(). This method is inherited by every HTML element node and can be used to query just *portions* of the DOM instead of the entire thing. It accepts a single argument, a case-insensitive string of the HTML tag you want to find. For example, to get a list of *all* the <div> tags on the page, you might use:

```
  document.getElementsByTagName("div");
```

Like getElementsByName(), this will return a NodeList collection. If no nodes are found, it will be a collection of zero elements.

You can also narrow down your search by using the method on a sub-node of the DOM. For example, suppose you have a reference to an HTML <table> node. You can get a collection of all the cells in the table by using getElementsByTagName() *on* that table node:

```
  var cellObjs = tableObj.getElementsByTagName("td");
```

In all modern browsers (and IE 6+), you can use a wildcard symbol (*) to get a collection of *all* the elements in a portion of the document. For example, to get a list of all the tagged elements in the table object, you can write:

```
  var allElements = tableObj.getElementsByTagName("*");
```

Like other NodeList's, the resulting array has a length property but none of the other features of the Array object.

getElementById

Maybe the most important DOM utility of all is document.getElementById(). I say *important* because it's the singularly most-popular DOM function of all. It uses the HTML id attribute to locate a *specific* DOM node out of all the other nodes in the document. If getElementById() can't *find* an element with the specified ID, it returns null. If it finds multiple elements with that ID, it returns the first one.

If you have an HTML element like this:

```
<img src="myLogo.gif" id="myImg">
```

You can instantly retrieve a reference to it from the document by writing:

```
var imgRef = document.getElementById('myImg');
```

Remember that IDs are case sensitive. Also, a common typo among developers is to capitalize the d at the end of getElementById. This will trigger a TypeError.