

JavaScript: The Basics

Instructor: Frank Stepanski

Overview

In this class, you will learn what the JavaScript language is, how it can be used, look at useful examples, and the essentials to give you a solid understanding of the JavaScript language.

Audience for this Class

This class is considered a “beginner” level class, therefore, I will assume no knowledge of any type of programming skill. I will assume you have some basic understanding of HTML/XHTML. You do need to know HTML/XHTML to take this class, but it would be helpful in your learning because all the coding examples are used within HTML pages (.html).

If you need extra help with learning XHTML please let me know and I will explain any code that is used in the lessons and examples.

Tools Needed

There is no specific tool to develop in JavaScript so you can use any free text or web editor like [Notepad++](#) or [HTML-Kit](#) or [Komodo Edit](#) or commercial products like [Adobe Dreamweaver](#) or [Microsoft Web Expression](#).

History of JavaScript

[Brendan Eich](#), who started working for Netscape in 1995, began developing a scripting language called LiveScript for the release of Netscape Navigator 2 in late 1995, with the intention of using it both in the browser and on the server (where it was to be called LiveWire). Netscape entered into a development alliance with Sun Microsystems to complete the implementation of LiveScript in time for release.

Just before Netscape Navigator 2 was officially released, Netscape changed LiveScript's name to JavaScript to capitalize on the buzz that Java was receiving from the Press. Because JavaScript and their browser were such a hit, Microsoft decided to put more resources into a competing browser named Internet Explorer.

Shortly after Netscape Navigator 3 was released, Microsoft introduced Internet Explorer 3 with a JavaScript implementation called [JScript](#) (so called to avoid any possible licensing issues with Netscape). This major step for Microsoft into the realm of web browsers in August 1996 represented a major step forward in the development of JavaScript as a language.

Microsoft's implementation of JavaScript meant that there were two different JavaScript versions floating around: JavaScript in Netscape Navigator, JScript in Internet Explorer.

Unlike C and many other programming languages, JavaScript had no standards governing its syntax or features, and the two different versions only highlighted this problem. With industry fears mounting, it was decided that the language must be standardized. In 1997, JavaScript 1.1 was submitted to the [European Computer Manufacturers Association](#) (Ecma) as a proposal.

[Technical Committee #39](#) (TC39) was assigned to “standardize the syntax and semantics of a general purpose, cross - platform, vendor - neutral scripting language”.

Made up of programmers from Netscape, Sun, Microsoft, Borland, and other companies with interest in the future of scripting, TC39 met for months to hammer out [ECMA - 262](#), a standard defining a new scripting language named [ECMAScript](#).

Though JavaScript and ECMAScript are often used synonymously, JavaScript is much more than just what is defined in ECMA - 262.

A complete JavaScript implementation is made up of the following three distinct parts:

1. The Core language (ECMAScript)
2. The Document Object Model (DOM)
3. The Browser Object Model (BOM)

ECMAScript

ECMAScript, the language defined in ECMA - 262, isn't tied to web browsers. In fact, the language has no methods for input or output whatsoever. ECMA - 262 defines this language as a base upon which more - robust scripting languages may be built. Web browsers are just one *host environment* in which an ECMAScript implementation may exist. A host environment provides the base implementation of ECMAScript as well as extensions to the language designed to interface with the environment itself.

ECMAScript is simply a description of a language implementing all of the facets ascribed in the specification. Other scripting languages such as ActionScript 3.0 for Adobe Flash are based on ECMAScript.

The five major web browsers (Internet Explorer, Firefox, Safari, Chrome, and Opera) all comply with the third edition of ECMA - 262.

The Document Object Model (DOM)

The *Document Object Model* (DOM) is an application programming interface (API) for XML that was extended for use in HTML. The DOM maps out an entire page as a hierarchy of nodes. Each part of an HTML or XML page is a type of a node containing different kinds of data.

By creating a tree to represent a document, the DOM allows developers an unprecedented level of control over its content and structure. Nodes can be removed, added, replaced, and modified easily by using the DOM API.

With the DOM, developers can alter the appearance and content of a web page without reloading it (i.e. the beginning of Ajax).

DOM: Support in Browsers

The DOM had been a standard for some time before web browsers started implementing it. Internet Explorer made its first attempt with version 5, but it didn't have any realistic DOM support until version 5.5, when it implemented most of DOM Level 1. Internet

Explorer hasn't introduced new DOM functionality in versions 6 and 7, though version 8 introduces some bug fixes.

For Netscape, no DOM support existed until Netscape 6 (Mozilla 0.6.0) was introduced. After Netscape 7, Mozilla switched its development efforts to the Firefox browser.

Firefox 3 supports all of Level 1, nearly all of Level 2, and some parts of Level 3.

DOM support became a huge priority for most browser vendors, and efforts have been ongoing to improve support with each release. Chrome 0.2+, Opera 9, and Safari 3 support all of DOM Level 1 and most of DOM Level 2. Internet Explorer now lags far behind the other three major browsers in DOM support, being stuck at a partial implementation of DOM Level 1.

Note: I will be talking more about how to use the DOM for dynamic page manipulation in my [JavaScript Intermediate: Unobtrusive Programming](#) class.

The Browser Object Model (BOM)

All modern browsers offer a version of the *Browser Object Model* (BOM) that allows access and manipulation of the browser window. Using the BOM, developers can interact with the browser outside of the context of its displayed page. What makes the BOM truly unique, and often problematic, is that it is the only part of a JavaScript implementation that has no related standard.

Primarily, the BOM deals with the browser window and frames, but generally any browser – specific extension to JavaScript is considered to be a part of the BOM. The following are some such extensions:

1. The capability to pop up new browser windows
2. The capability to move, resize, and close browser windows
3. The navigator object, which provides detailed information about the browser the location object, which gives information about the page loaded in the browser.
4. The screen object, which gives information about the user's screen resolution.
5. Support for cookies.
6. Custom objects such as XMLHttpRequest and Internet Explorer's ActiveXObject

Because no standards exist for the BOM, each browser has its own implementation. There are some *defacto* standards, such as having a window object and a navigator object, but each browser defines its own properties and methods for these and other objects.

Note: I will cover the BOM in a later lesson.

JavaScript Versions

Mozilla (creator of Firefox), a descendant from the original Netscape, is the only browser vendor that has continued the original JavaScript version which is a numbering sequence.

Version	Release date	Equivalent to	Netscape Navigator	Mozilla Firefox	Internet Explorer	Opera	Safari	Google Chrome
1.0	March 1996		2.0		3.0			
1.1	August 1996		3.0					
1.2	June 1997		4.0-4.05					
1.3	October 1998	ECMA-262 1 st edition / ECMA-262 2 nd edition	4.06-4.7x		4.0			
1.4			Netscape Server					
1.5	November 2000	ECMA-262 3 rd edition	6.0	1.0	5.5 (JScript 5.5), 6 (JScript 5.6), 7 (JScript 5.7), 8 (JScript 6)	6.0, 7.0, 8.0, 9.0		
1.6	November 2005	1.5 + Array extras + Array and String generics + E4X		1.5			3.0, 3.1	
1.7	October 2006	1.6 + Pythonic generators + Iterators + let		2.0			3.2, 4.0	1.0
1.8	June 2008	1.7 + Generator expressions + Expression closures		3.0				

Figure 1 – Versions of JavaScript: <http://en.wikipedia.org/wiki/JavaScript>

The numbering scheme is based on the idea that Firefox 4 will feature JavaScript 2.0, and each increment in the version number prior to that point indicates how close the JavaScript implementation is to the 2.0 proposal. Though this was the original plan, it is unclear if Mozilla will continue along this path given the popularity of the ECMAScript 3.1 proposal.

The <script> Tag and Your First Simple JavaScript Program

Inserting JavaScript into a web page is much like inserting any other HTML content; you use tags to mark the start and end of your script code. The tag used to do this is the <script> tag. This tells the browser that the following chunk of text, bounded by the closing </script> tag, is not HTML to be displayed, but rather script code to be processed.

The chunk of code surrounded by the <script> and </script> tags is called a *script block*.

Basically, when the browser spots <script> tags, instead of trying to display the contained text to the user, it uses the browser's built-in JavaScript interpreter to run the code's instructions. Of course, the code might give instructions about changes to the way the page is displayed or what is shown in the page, but the text of the code itself is never shown to the user.

You can put the <script> tags inside the header (between the <head> and </head> tags), or inside the body (between the <body> and </body> tags) of the HTML page. However, although you can put them outside these areas—for example, before the <html> tag or after the </html> tag—this is not permitted in the web standards and so is considered bad practice.

The <script> tag has a number of attributes, but the most important one is the type attribute. You can tell the browser which scripting language to expect so that it knows how to process that language. Your opening script tag will look like this:

```
<script type="text/javascript">
```

Including the type attribute is good practice, but within a web page it can be left off. Browsers such as IE and Firefox use JavaScript as their default script language.

This means that if the browser encounters a <script> tag with no type attribute set, it assumes that the script block is written in JavaScript.

However, use of the type attribute is specified as mandatory by [W3C](#) (the World Wide Web Consortium), which sets the standards for HTML and XHTML.

Okay, let's take a look at the first page containing JavaScript code.

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>

<script type="text/javascript">
    document.bgColor = "RED";
</script>

</body>
</html>
```

Save the page as **example1.html**. Now load it into your web browser. You should see a red web page with the text Paragraph 1 in the top-left corner.

But wait didn't you set the <body> tag's BGCOLOR attribute to white?

Okay, let's look at what's going on here.

The page is contained within <html> and </html> tags. This block contains a <body> element. When you define the opening <body> tag, you use HTML to set the page's background color to white.

```
<BODY BGCOLOR="WHITE">
```

Then you let the browser know that your next lines of code are JavaScript code by using the <script> start tag.

```
<script type="text/javascript">
```

Everything from here until the close tag, </script>, is JavaScript and is treated as such by the browser. Within this script block, you use JavaScript to set the document's background color to red.

```
document.bgColor = "RED";
```

What you might call the *page* is known as the *document* for the purpose of scripting in a web page. The document has lots of properties, including its background color, bgColor.

You can reference properties of the document by writing document, followed by a dot, followed by the property name.

Note: Don't worry about the use of document at the moment; we'll look at this later in depth in a later lesson.

The preceding line of code is an example of a JavaScript *statement*. Every line of code between the <script> and </script> tags is called a statement, although some statements may run on to more than one line.

You'll also see that there's a semicolon (;) at the end of the line. You use a semicolon in JavaScript to indicate the end of a statement. In practice, JavaScript is very relaxed about the need for semicolons, and when you start a new line, JavaScript will usually be able to work out whether you mean to start a new line of code. However, for good coding practice, you should use a semicolon at the end of statements of code, and a single JavaScript statement should fit onto one line rather than continue on to two or more lines.

Finally, to tell the browser to stop interpreting your text as JavaScript and start interpreting it as HTML, you use the script close tag:

```
</script>
```

You've now looked at how the code works, but you haven't looked at the order in which it works.

When the browser loads in the web page, the browser goes through it, rendering it tag by tag from top to bottom of the page. This process is called *parsing*. The web browser starts at the top of the page and works its way down to the bottom of the page. The browser comes to the <body> tag first and sets the document's background to white. Then it continues parsing the page. When it comes to the JavaScript code, it is instructed to change the document's background to red.

Note: For the purpose of teaching you how JavaScript works within a web page, I sometimes will be using seldom used HTML attributes. In current web design practices these seldom used (or even deprecated) HTML tags are now replaced by Cascading Style Sheets (CSS). I am using these tags as part of my JavaScript explanations and should not normally be used for current web design practices. ☺

Let's extend the previous example to demonstrate the parsing of a web page.

Type the following into your text editor:

```
<html>
<body bgcolor="WHITE">
<p>Paragraph 1</p>

<script type="text/javascript">
// Script block 1
alert("First Script Block");
</script>

<p>Paragraph 2</p>

<script type="text/javascript">
// Script block 2
document.bgColor = "RED";
alert("Second Script Block");
</script>

<p>Paragraph 3</p>
</body>
</html>
```

Save the file as **example2.html** and then load it into your browser. When you load the page you should see the first paragraph, Paragraph 1, followed by a message box displayed by the first script block. The browser halts its parsing until you click the OK button.

The page background is white, as set in the <body> tag, and only the first paragraph is currently displayed. Click the OK button, and the parsing continues. The browser displays the second paragraph, and the second script block is reached, which changes the background color to red. Another message box is displayed by the second script block

Click OK, and again the parsing continues, with the third paragraph, Paragraph 3, being displayed.

