

JavaScript Programming: From Basics to DOM

Instructor: Frank Stepanski

Overview

In this class, you will learn what the JavaScript language is, how it can be used, look at useful examples, how to develop code focusing on unobtrusiveness and progressive enhancement, manipulating the DOM, and much more. After taking this class you should have a very solid understanding of how to program in JavaScript.

Audience for this Class

This class starts at the “beginner” level, therefore, I will assume no knowledge of any type of programming skill. I will assume you have some 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).

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.

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>
```

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).

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.

How It Works

The first part of the page is the same as in our earlier example. The background color for the page is set to white in the definition of the <body> tag, and then a paragraph is written to the page.

```

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

```

The first new section is contained in the first script block.

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

This script block contains two lines, both of which are new to you. The first line—

```
// Script block 1
```

is just a *comment*, solely for your benefit. The browser recognizes anything on a line after a double forward slash (//) to be a comment and does not do anything with it. It is useful for you as a programmer because you can add explanations to your code that make it easier to remember what you were doing when you come back to your code later.

The [alert\(\)](#) function in the second line of code is also new to you. Before learning what it does, you need to know what a *function* is.

A function is a piece of JavaScript code that you can use to do certain tasks. A function takes some information, processes it, and gives you a result.

In particular, the alert() function enables you to alert or inform the user about something by displaying a message box. The message to be given in the message box is specified inside the parentheses of the alert() function and is known as the function's *parameter*.

The message box displayed by the alert() function is *modal* which means that the message box won't go away until the user closes it by clicking the OK button. In fact, parsing of the page stops at the line where the alert() function is used and doesn't restart until the user closes the message box.

This is quite useful for this example, because it enables you to demonstrate the results of what has been parsed so far: The page color has been set to white, and the first paragraph has been displayed. When you click OK, the browser carries on parsing down the page through the following lines:

```
<p>Paragraph 2</p>  
<script type="text/javascript">
```

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

The second paragraph is displayed, and the second block of JavaScript is run. The first line of the script block code is another comment, so the browser ignores this. You saw the second line of the script code in the previous example—it changes the background color of the page to red. The third line of code is the `alert()` function, which displays the second message box. Parsing is brought to a halt until you close the message box by clicking OK.

When you close the message box, the browser moves on to the next lines of code in the page, displaying the third paragraph and finally ending the web page.

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

Another important point raised by this example is the difference between setting properties of the page, such as background color, via HTML and doing the same thing using JavaScript. The method of setting properties using HTML is *static*. A value can be set only once and never changed again by means of HTML. Setting properties using JavaScript enables you to dynamically change their values.

By the term *dynamic*, we are referring to something that can be changed and whose value or appearance is not set in stone.

This example is just that, an example. In practice if you wanted the page background to be red, you would set the `<body>` tag's `BGCOLOR` attribute to "RED", and not use JavaScript at all. Where you would want to use JavaScript is where you want to add some sort of intelligence or logic to the page.

Note: Using CSS is the most practical way to change the background color of a page. (i.e. `body {background: color}`)

JavaScript: Case Sensitive

Everything is case-sensitive: variables, function names, and operators are all case-sensitive, meaning that a variable named *test* is different from a variable named *Test*.

JavaScript: Comments

A single - line comment begins with two forward - slash characters, such as this:

```
//single line comment
```

A block comment begins with a forward - slash and asterisk (*/**), and ends with the opposite (**/*), as in this example:

```
/*  
 * This is a multi-line  
 * Comment  
*/
```

Note: Even though the second and third lines contain an asterisk, these are not necessary and are added purely for readability (this is the format preferred in enterprise applications).

JavaScript: Variables

Variables are loosely typed, meaning that a variable can hold any type of data.

Every variable is simply a named placeholder for a value. To define a variable, use the `var` operator (note that `var` is a keyword) followed by the variable name (an identifier, as described earlier), like this:

```
var message;
```

This code defines a variable named *message* that can be used to hold any value (without initialization, it holds the special value `undefined`).

```
var message = "hi";
```

Here, `message` is defined to hold a string value of `"hi"`. Doing this initialization doesn't mark the variable as being a string type; it is simply the assignment of a value to the variable. It is still possible to not only change the value stored in the variable, but also to change the type of value, such as this:

```
var message = "hi";  
message = 100; //legal, but not recommended
```

In this example, the variable `message` is first defined as having the string value `"hi"` and then overwritten with the numeric value `100`. Though it's not recommended to switch the data type that a variable works with, it is completely valid.

Also, you can't use certain names and characters for your variable names. Names you can't use are called *reserved* words. Reserved words are words that JavaScript keeps for its own use, for example the word `var` or the word `with`. Certain characters are also forbidden in variable names; for example, the ampersand (`&`) and the percent sign (`%`).

You are allowed to use numbers in your variable names, but the names must not begin with numbers. So `101myVariable` is not okay, but `myVariable101` is.

Invalid names include:

```
with  
99variables  
my%Variable  
theGood&theBad
```

Valid names include:

```
myVariable99  
myPercent_Variable  
the_Good_and_the_Bad
```

JavaScript: Data Types

There are five simple data types (also called *primitive types*) in JavaScript:

Undefined, Null, Boolean, Number, and String.

There is also one complex data type called Object, which is an unordered list of name - value pairs. Because there is no way to define your own data types, all values can be represented as one of these six.

Because JavaScript is loosely typed, there needs to be a way to determine the data type of a given variable. The [typeof](#) operator provides that information. Using the typeof operator on a value returns one of the following strings:

“undefined” if the value is undefined
“boolean” if the value is a Boolean
“string” if the value is a string
“number” if the value is a number
“object” if the value is an object or null
“function” if the value is a function

The typeof operator is called like this:

```
var message = "some string";  
alert(typeof message); // "string"  
alert(typeof(message)); // "string"  
alert(typeof 95); // "number"
```

In this example, both a variable (message) and a numeric literal are passed into the typeof operator.

Note: Technically, functions are considered objects and don't represent another data type.

JavaScript: Numerical Data

Numerical data come in two forms:

1. Whole numbers, such as 145, which are also known as *integers*. These numbers can be positive or negative and can span a very wide range in JavaScript: -253 to 253.
2. Fractional numbers, such as 1.234, which are also known as *floating-point* numbers. Like integers, they can be positive or negative, and they also have a massive range.

JavaScript: Text Data

Another term for one or more characters of text is a *string*. You tell JavaScript that text is to be treated as text and not as code simply by enclosing it inside quote marks (“). For example, “Hello World” and “A” are examples of strings that JavaScript will recognize.

You can also use the single quote marks (‘), so ‘Hello World’ and ‘A’ are also examples of strings that JavaScript will recognize.

However, you must end the string with the same quote mark that you started it with. Therefore, “A’ is not a valid JavaScript string, and neither is ‘Hello World”.

What if you want a string with a single quote mark in the middle, say a string like Peter O’Toole? If you enclose it in double quotes, you’ll be fine, so “Peter O’Toole” is recognized by JavaScript.

However, ‘Peter O’Toole’ will produce an error. This is because JavaScript thinks that your text string is Peter O (that is, it treats the middle single quote as marking the end of the string) and falls over wondering what the Toole’ is.

Another way around this is to tell JavaScript that the middle ‘ is part of the text and is not indicating the end of the string. You do this by using the backslash character (\), which has special meaning in JavaScript and is referred to as an *escape character*. The backslash tells the browser that the next character is not the end of the string, but part of the text. So ‘Peter O\’Toole’ will work as planned.

What if you want to use a double quote inside a string enclosed in double quotes? Well, everything just said about the single quote still applies. So `'Hello "Paul"'` works, but `"Hello "Paul""` won't. However, `"Hello \"Paul\""` will also work.

JavaScript has a lot of other special characters, which can't be typed in but can be represented using the escape character in conjunction with other characters to create *escape sequences*. These work much the same as in HTML. For example, more than one space in a row is ignored in HTML, so a space is represented by the term ` `.

Similarly, in JavaScript there are instances where you can't use a character directly but must use an escape sequence.

The following table details some of the more useful escape sequences:

<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash

Setting Up Your Browser for Errors

Although your code has been fairly simple so far, it is still possible to make errors when typing it in. As you start to look at more complex and detailed code, this will become more and more of a problem. So, before we continue on other more detailed topics it seems like a good point to discuss how to ensure that any errors that arise in your code are shown to you by the browser, so that you can go and correct them.

When you are surfing other people's web sites, you probably won't be interested in seeing when there are errors in their code. In this situation, it's tempting to find a way of switching off the display of error dialog boxes in the browser.

However, as JavaScript programmers, we want to know all the gory details about errors in our own web pages; that way we can fix them before someone else spots them. It's

important, therefore, to make sure the browsers we use to test our web sites are configured correctly to show errors and their details.

Displaying Errors in Firefox

Firefox keeps quiet about your errors, so if things go wrong you won't see any pop-up boxes warning you or alarm bells going off. However, one of its developer tools is the Error Console, which contains details of any JavaScript problems on your page. It also reports other problems, such as invalid CSS.

To view this console, go to the Tools menu and select Error Console. The Error Console then pops open in its own separate window, as shown in Figure 2. While you're developing your JavaScript code, it's as well to leave the Console open. At the moment, it is probably blank.

Displaying Errors in Internet Explorer

Normally, IE will by default display JavaScript errors using dialog boxes. However, it is possible to turn off the displaying of such errors, in which case you need to follow a few simple steps to re-enable error displaying. First open up Internet Explorer and select the Internet Options menu from the Tools menu.

In the dialog box that appears, select the Advanced tab. Under Browsing, make sure the Disable script debugging (Other) check box is cleared and that the Display a notification about every script error check box is selected (Figure 3).

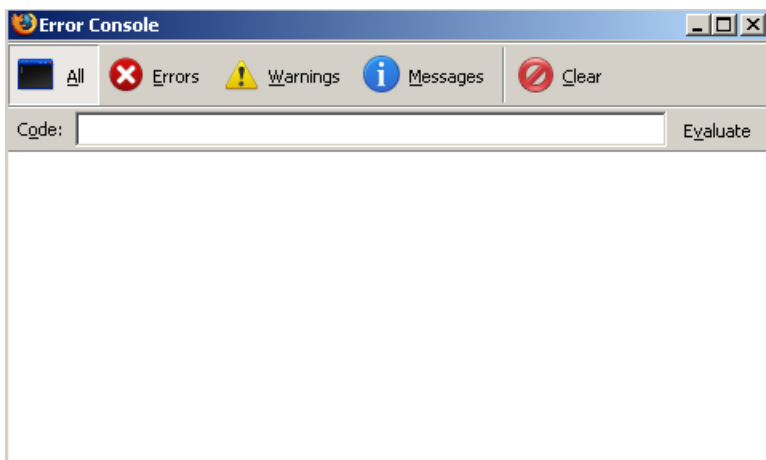


Figure 2 – Firefox Error console

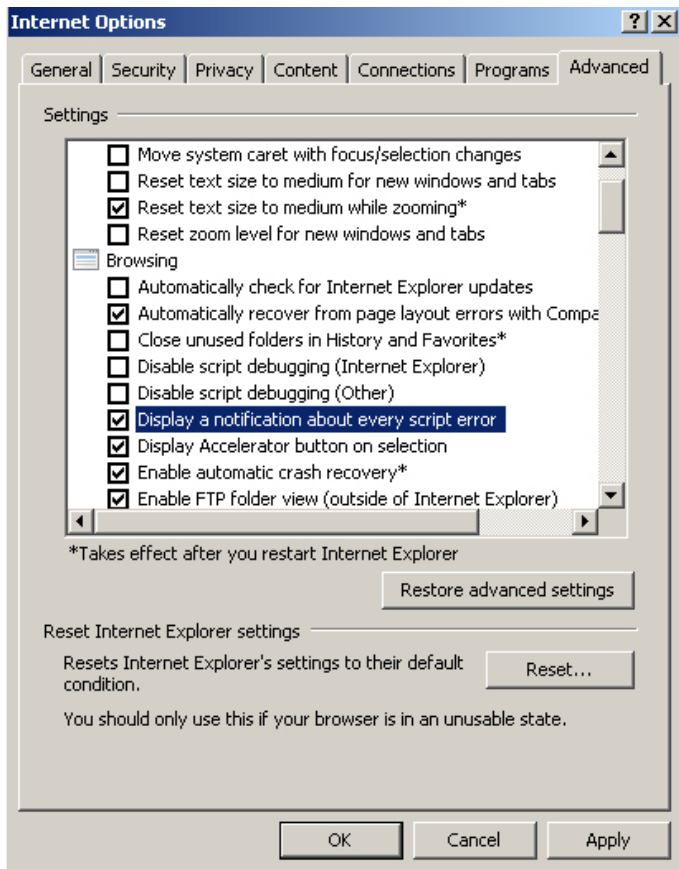


Figure 3 – Internet Explorer Advanced Settings tab

What Happens When You Get an Error

As mentioned earlier, the use of a reserved word in a variable name will result in a JavaScript error in the browser. However, the error message displayed may not be instantly helpful since it may not indicate that you've used a reserved word in declaring your variables.

Let's look at the sort of error messages you might see in this situation. Note that these error messages can also be produced by other mistakes not related to variable naming, which can get confusing at times.

Let's assume that you try to define a variable called with like this:

```
var with;
```

The word with is reserved in JavaScript. What errors will you see?

In Firefox you see nothing unless you open the Error console, in which case you see something like what is shown in Figure 4.

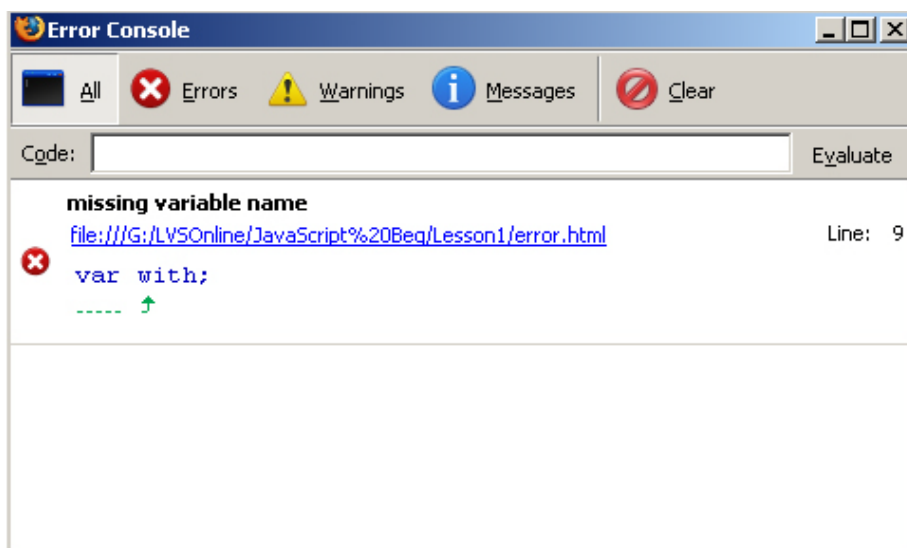


Figure 4 – Error message displayed in Firefox Error Console

In Internet Explorer, as long as you have the display errors enabled, as discussed in the section above, the sort of message you can expect to see is shown in Figure 5.

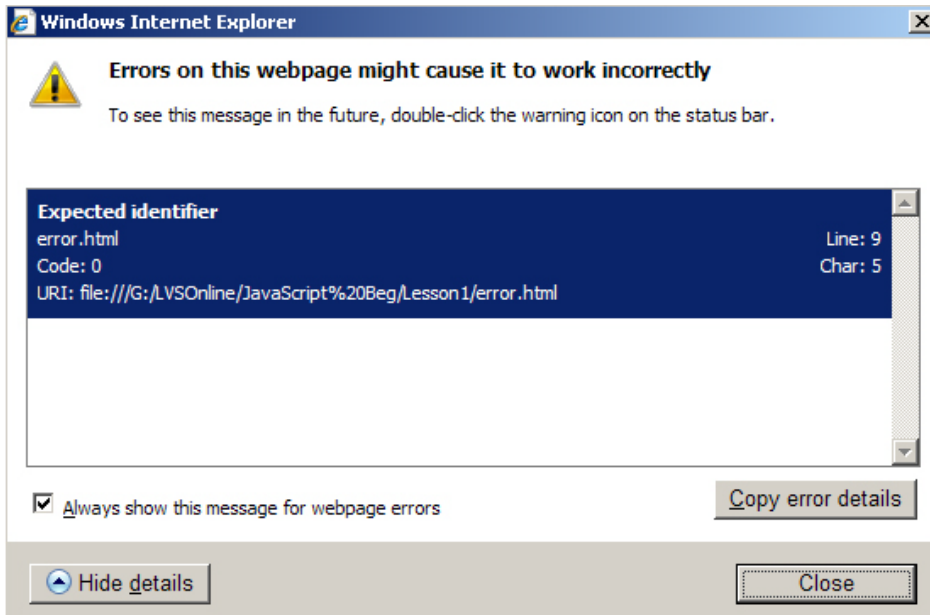


Figure 5 – Error message displayed in Internet Explorer (version 7 or 8)

Using Data — Calculations and Basic String Manipulation

Variables enable you to temporarily hold information that you can use for processing in mathematical calculations, in building up text messages, or in processing words that the user has entered.

JavaScript has a range of basic mathematical capabilities, such as addition, subtraction, multiplication, and division. Each of the basic math functions is represented by a symbol: plus (+), minus (-), star (*), and forward slash (/), respectively. These symbols are called *operators* because they operate on the values you give them. In other words, they perform some calculation or operation and return a result to us. You can use the results of these calculations almost anywhere you'd use a number or a variable.

Imagine you were calculating the total value of items on a shopping list. You could write this:

Total cost of shopping = 10 + 5 + 5

or, if you actually calculate the sum, it's

Total cost of shopping = 20

Now let's see how to do this in JavaScript. In actual fact, it is very similar except that you need to use a variable to store the final total.

```
var TotalCostOfShopping;
```

```
TotalCostOfShopping = 10 + 5 + 5;
```

```
alert(TotalCostOfShopping);
```

Fahrenheit to Centigrade

Let's look at a more complex but very common example:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">
// Equation is °C = 5/9 (°F - 32).

var degFahren = prompt("Enter the degrees in Fahrenheit",50);
var degCent;
degCent = 5/9 * (degFahren - 32);
alert(degCent);

</script>

</body>
</html>
```

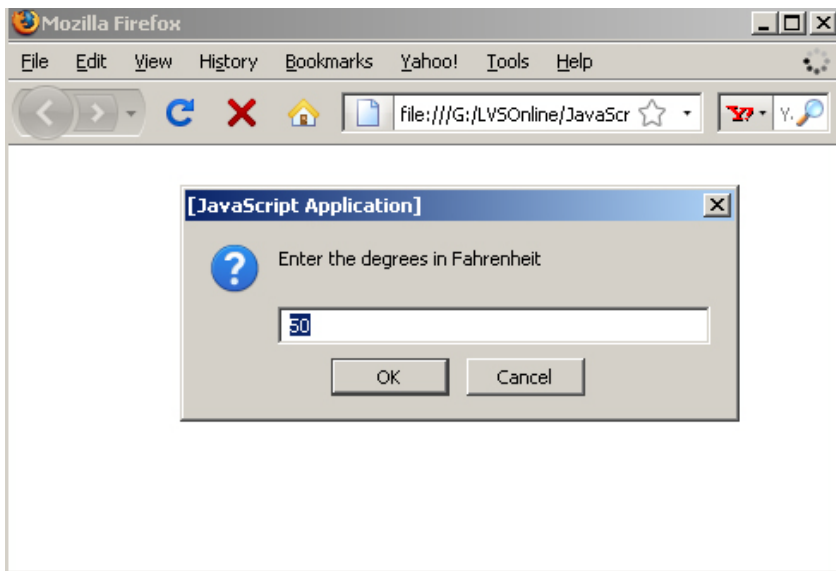


Figure 6 – Fahrenheit to Centigrade script

If you load the page into your browser, you should see a prompt box, like that shown in Figure 6, which asks you to enter the degrees in Fahrenheit to be converted. The value 50 is already filled in by default.

If you leave it at 50 and click OK, an alert box with the number 10 in it appears. This represents 50 degrees Fahrenheit converted to centigrade. Reload the page and try changing the value in the prompt box to see what results you get.

For example, change the value to 32 and reload the page. This time you should see 0 appear in the box.

Note: As it's still a fairly simple example, there's no checking of data input so it'll let you enter abc as the degrees Fahrenheit.

How It Works

The first line of the script block is a comment since it starts with two forward slashes (`//`).

Then your variables are declared.

```
var degFahren = prompt("Enter the degrees in Fahrenheit",50);  
var degCent;
```

Instead of initializing the `degFahren` variable to a literal value, you get a value from the user using the [prompt\(\)](#) function. The `prompt()` function works in a similar way to an `alert()` function, except that as well as displaying a message, it also contains a text box in which the user can enter a value. It is this value that will be stored inside the `degFahren` variable. The value returned is a text string but this will be implicitly converted by JavaScript to a number when you use it as a number.

You pass two pieces of information to the `prompt()` function:

1. The text to be displayed—usually a question that prompts the user for input.
2. The default value that is contained in the input box when the prompt dialog box first appears.

These two pieces of information must be specified in the given order and separated by a comma. If you don't want a default value to be contained in the input box when the prompt box opens, you should use an empty string ("") for the second piece of information.

Next in the script block comes the equation represented in JavaScript. You store the result of the equation in the `degCent` variable.

```
degCent = 5/9 * (degFahren - 32);
```

The calculation of the expression on the right-hand side of the equals sign raises a number of important points. First, just as in math, the JavaScript equation is read from left to right, at least for the basic math functions like +, -, and so on. Secondly, just as there is precedence in math, there is in JavaScript.

Starting from the left, first JavaScript works out $5/9 = .5556$ (approximately). Then it comes to the multiplication, but wait . . . the last bit of our equation, `degFahren - 32`, is in parentheses. This raises the order of precedence and causes JavaScript to calculate the result of `degFahren - 32` before doing the multiplication.

Finally, in your script block, you display the answer using the `alert()` function.

```
alert(degCent);
```

Basic String Operations

One thing you'll find yourself doing again and again in JavaScript is joining two strings together to make one string—a process that's termed *concatenation*.

For example, you may want to concatenate the two strings "Hello" and "Paul" to make the string "Hello Paul". So how do you concatenate? Easy!

Use the + operator. Recall that when applied to numbers, the + operator adds them up, but when used in the context of two strings, it joins them together.

```
var concatString = "Hello " + "Paul";
```

The string now stored in the variable `concatString` is “Hello Paul”. Notice that the last character of the string “Hello” is a space — if you left this out, your concatenated string would be “HelloPaul”.

Arrays

An array is similar to a normal variable, in that you can use it to hold any type of data. However, it has one important difference, which you’ll see below.

As you have already seen, a normal variable can only hold one piece of data at a time. For example, you can set `myVariable` to be equal to 25 like so:

```
myVariable = 25;
```

and then go and set it to something else, say 35:

```
myVariable = 35;
```

However, when you set the variable to 35, the first value of 25 is lost. The variable `myVariable` now holds just the number 35. The difference between such a normal variable and an array is that an array can hold *more than one* item of data at the same time.

For example, you could use an array with the name `myArray` to store both the numbers 25 and 35. Each place where a piece of data can be stored in an array is called an *element*. How do you distinguish between these two pieces of data in an array? You give each piece of data an *index* value. To refer to that piece of data you enclose its index value in square brackets after the name of the array.

An array called `myArray` containing the data 25 and 35 could be illustrated as:

Element name	Value
<code>myArray[0]</code>	25
<code>myArray[1]</code>	35

Notice that the index values start at 0 and not 1.

Arrays can be very useful since you can store as many or as few items of data in an array as you want. Also, you don't have to say up front how many pieces of data you want to store in an array, though you can if you wish.

So how do you create an array? This is slightly different from declaring a normal variable. To create a new array, you need to declare a variable name and tell JavaScript that you want it to be a new array using the [new](#) keyword and the [Array\(\)](#) function.

For example, the array `myArray` could be defined like this:

```
var myArray = new Array();
```

Note that, as with everything in JavaScript, the code is case-sensitive, so if you type `array()` rather than `Array()`, the code won't work. As with normal variables, you can also declare your variable first, and then tell JavaScript you want it to be an array.

```
var myArray;  
myArray = new Array();
```

I mentioned earlier that you can say up front how many elements the array will hold if you want to, although this is not necessary. You do this by putting the number of elements you want to specify between the parentheses after `Array`.

For example, to create an array that will hold **six** elements, you write the following:

```
var myArray = new Array(6);
```

You have seen how to declare a new array, but how do you store your pieces of data inside it? You can do this when you define your array by including your data inside the parentheses, with each piece of data separated by a comma.

For example:

```
var myArray = new Array("Paul",345,"John",112,"Bob",99);
```

Here the first item of data, "Paul", will be put in the array with an index of 0. The next piece of data, 345, will be put in the array with an index of 1, and so on. This means that

the element with the name `myArray[0]` contains the value “Paul”, the element with the name `myArray[1]` contains the value 345, and so on.

This leads to another way of declaring data in an array. You could write the preceding line like this:

```
var myArray = new Array();
myArray[0] = “Paul”;
myArray[1] = 345;
myArray[2] = “John”;
myArray[3] = 112;
myArray[4] = “Bob”;
myArray[5] = 99;
```

You use each element name as you would a variable, assigning them with values.

Try It Out:

```
<html>
<body>

<script language=“JavaScript” type=“text/javascript”>
var myArray = new Array();
myArray[0] = “Bob”;
myArray[1] = “Pete”;
myArray[2] = “Paul”;
document.write(“myArray[0] = “ + myArray[0] + “<br/>”);
document.write(“myArray[2] = “ + myArray[2] + “<br/>”);
document.write(“myArray[1] = “ + myArray[1] + “<br/>”);
myArray[1] = “Mike”;
document.write(“myArray[1] changed to “ + myArray[1]);
</script>

</body>
</html>
```

Decision Making — The if and switch Statements

All programming languages enable you to make decisions—that is, they enable the program to follow a certain course of action depending on whether a particular *condition* is met. This is what gives programming languages their intelligence.

Decision making also has its own operators, which enable you to test conditions.

Comparison operators, just like the mathematical operators you saw in the last chapter, have a left-hand side (LHS) and a righthand side (RHS), and the comparison is made between the two. The technical terms for these are the *left operand* and the *right operand*.

For example, the less-than operator, with the symbol $<$, is a comparison operator. You could write $23 < 45$, which translates as “Is 23 less than 45?” Here, the answer would be true.

Operator Symbol	Purpose
==	Tests if LHS is equal to RHS
<	Tests if LHS is less than RHS
>	Tests if LHS is greater than RHS
<=	Tests if LHS is less than or equal to RHS
>=	Tests if LHS is greater than or equal to RHS
!=	Tests if LHS is not equal to RHS

Assignment versus Comparison

One very important point to mention is the ease with which the assignment operator ($=$) and the comparison operator ($==$) can be mixed up. Remember that the $=$ operator assigns a value to a variable and that the $==$ operator compares the value of two variables. Even when you have this idea clear, it’s amazingly easy to put one equals sign where you meant to put two.

The if Statement

The [if statement](#) is one you'll find yourself using in almost every program that is more than a couple of lines long. It works very much as it does in the English language.

For example, you might say in English, "If the room temperature is more than 80 degrees Fahrenheit, then I'll turn the air conditioning on." In JavaScript, this would translate into something like this:

```
if (roomTemperature > 80)
{
    roomTemperature = roomTemperature - 10;
}
```

Notice that the test condition is placed in parentheses and follows the if keyword. Also, note that there is no semicolon at the end of this line. The code to be executed if the condition is true is placed in curly braces on the line after the condition.

The curly braces, {}, have a special purpose in JavaScript: They mark out a *block* of code. Marking out lines of code as belonging to a single block means that JavaScript will treat them all as one piece of code.

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var degFahren = Number(prompt("Enter the degrees Fahrenheit",32));
var degCent;
degCent = 5/9 * (degFahren - 32);
document.write(degFahren + "\xB0 Fahrenheit is " + degCent +
"\xB0 centigrade<br/>");
if (degCent < 0)
{
    document.write("That's below the freezing point of water");
}
if (degCent == 100)
{
    document.write("That's the boiling point of water");
}
```

```
</script>

</body>
</html>
```

Load the page into your browser and enter 32 into the prompt box for the Fahrenheit value to be converted. With a value of 32, neither of the if statement's conditions will be true, so the only line written in the page will be:

32° Fahrenheit is 0° centigrade

Try out some other values and see the result. ☺

Logical Operators

You should have a general idea of how to use conditions in if statements now, but how do you use a condition such as “Is degFahren greater than zero, but less than 100?”

There are two conditions to test here. You need to test whether degFahren is greater than zero *and* whether degFahren is less than 100. JavaScript enables you to use such multiple conditions. To do this you need to learn about three more operators, the logical operators AND, OR, and NOT. The symbols for these are listed in the following table.

Operator	Symbol
AND	&&
OR	
NOT	!

Notice that the AND and OR operators are *two* symbols repeated: && and ||. If you type just one symbol, & or |, strange things will happen because these are special operators called *bitwise operators* used in binary operations—for logical operations you must always use two.

AND

The AND operator works very much as it does in English. For example, you might say, “If I feel cold *and* I have a coat, then I’ll put my coat on.” Here, the left-hand side of the “and” word is “Do I feel cold?” and this can be evaluated as true or false. The right-hand side is “Do I have a coat?” which again is evaluated to either true or false. If the left-hand side is true (I am cold) *and* the right-hand side is true (I do have a coat), then you put your coat on.

OR

Just like AND, OR also works much as it does in English. For example, you might say that if it is raining *or* if it is snowing, then you’ll take an umbrella. If either of the conditions “it is raining” or “it is snowing” is true, then you will take an umbrella. Again, just like AND, the OR operator acts on two Boolean values (one from its left-hand side and one from its right-hand side) and returns another Boolean value.

NOT

In English, we might say, “If I’m *not* hot, then I’ll eat soup.” The condition being evaluated is whether we’re hot. The result is true or false, but in this example we act (eat soup) if the result is false.

To use the NOT operator, you put the condition you want reversed in parentheses and put the ! symbol in front of the parentheses.

For example:

```
if (!(degCent < 100))  
{  
    // Some code  
}
```

Multiple Conditions Inside an if Statement

Using nested if statements, your code would be:

```
if (degCent < 100)
{
    if (degCent > 0)
    {
        document.write("degCent is between 0 and 100");
    }
}
```

This would work, but it's a little verbose and can be quite confusing. JavaScript offers a better alternative —using multiple conditions inside the condition part of the if statement. The multiple conditions are strung together with the logical operators you just looked at.

So the preceding code could be rewritten like this:

```
if (degCent > 0 && degCent < 100)
{
    document.write("degCent is between 0 and 100");
}
```

The if statement's condition first evaluates whether degCent is greater than zero. If that is true, the code goes on to evaluate whether degCent is less than 100. Only if both of these conditions are true will the document.write() code line execute.

else and else if

Imagine a situation where you want some code to execute if a certain condition is true, and some other code to execute if it is false. You can achieve this by having two if statements, as shown in the following example:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}
```

```
if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write( "myAge is NOT between 0 and 10" );
}
```

The first if statement tests whether myAge is between 0 and 10, and the second for the situation where myAge is not between 0 and 10. However, JavaScript provides an easier way of achieving this: with an else statement. Again, the use of the word else is similar to its use in the English language. You might say, "If it is raining, I will take an umbrella; otherwise I will take a sun hat."

In JavaScript you can say if the condition is true, then execute one block of code; else execute an alternative block. Rewriting the preceding code using this technique, you would have the following:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write( "myAge is between 0 and 10" );
}
else
{
    document.write( "myAge is NOT between 0 and 10" );
}
```

Writing the code like this makes it simpler and therefore easier to read.

The switch Statement

You saw earlier how the if and else if statements could be used for checking various conditions; if the first condition is not valid, then another is checked, and another, and so on. However, when you want to check the value of **a particular variable** for a large number of possible values, there is a more efficient alternative, namely the [switch](#) statement.

Try It Out

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var secretNumber = prompt("Pick a number between 1 and 5:", "");
secretNumber = parseInt(secretNumber);

switch (secretNumber)
{
    case 1:
        document.write("Too low!");
        break;
    case 2:
        document.write("Too low!");
        break;
    case 3:
        document.write("You guessed the secret number!");
        break;
    case 4:
        document.write("Too high!");
        break;
    case 5:
        document.write("Too high!");
        break;
    default:
        document.write("You did not enter a number between 1 and 5.");
        break;
}

document.write("<br/>Execution continues here");
</script>

</body>
</html>
```

How it Works

First you declare the variable `secretNumber` and set it to the value entered by the user via the prompt box. Note that you use the `parseInt()` function to convert the string that is returned from `prompt()` to an integer value.

```
var secretNumber = prompt("Pick a number between 1 and 5:", "");  
secretNumber = parseInt(secretNumber);
```

Next you create the start of the switch statement.

```
switch (secretNumber)  
{
```

The expression in parentheses is simply the variable `secretNumber`, and it's this number that the case statements will be compared against. You specify the block of code encompassing the case statements using curly braces. Each case statement checks one of the numbers between 1 and 5, because this is what you have specified to the user that she should enter. The first simply outputs a message that the number she has entered is too low.

```
case 1:  
document.write("Too low!");  
break;
```

The second case statement, for the value 2, has the same message, so the code is not repeated here. The third case statement lets the user know that she has guessed correctly.

```
case 3:  
document.write("You guessed the secret number!");  
break;
```

Finally, the fourth and fifth case statements output a message that the number the user has entered is too high.

```
case 4:  
document.write("Too high!");  
break;
```

You do need to add a default case in this example, since the user might very well (despite the instructions) enter a number that is not between 1 and 5, or even perhaps a letter. In this case you add a message to let the user know that there is a problem.

```
default:  
document.write("You did not enter a number between 1 and 5.");  
break;
```

A default statement is also very useful for picking up bugs—if you have coded some of the case statements incorrectly, you will pick that up very quickly if you see the default code being run when it shouldn't be. You finally have added the closing brace indicating the end of the switch statement. After this you output a line to indicate where the execution continues.

```
}  
document.write("<br/>Execution continues here");
```

Note: Each case statement ends with a `break` statement. This is important to ensure that execution of the code moves to the line after the end of the switch statement. If you forget to include this, you could end up executing the code for each case following the case that matches.

Additional Class Resources

JavaScript Overview:

https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/JavaScript_Overview

var keyword:

https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Statements/var

Variables: https://developer.mozilla.org/En/Core_JavaScript_1.5_Guide/Variables

Comparison operators:

https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Operators/Comparison_Operators

If statement:

https://developer.mozilla.org/En/Core_JavaScript_1.5_Reference/Statements/If...else

Switch statement:

https://developer.mozilla.org/En/Core_JavaScript_1.5_Reference/Statements/Switch

Assignment for Lesson 1

1. Review the lesson.
2. Practice writing simple scripts based upon the examples in this lesson.
3. Write a script that does the following:
 - a. Create a variable that stores your first name.
 - b. Display that variable in an alert box.
 - c. Take that variable and have it store your full name (first and last).
 - d. Display that variable in an alert box.
4. Review the code and correct the errors:

```
var userAge = prompt("Please enter your age");

if (userAge = 0);
{
    alert("So you're a baby!");
}
else if ( userAge < 0 / userAge > 200)
    alert("I think you may be lying about your age");
else
{
    alert("That's a good age");
}
```

5. Post on the class discussion board that your Lesson 1 is ready with a direct link to your page. I will respond with a review on the class discussion board.

Copyright 2011 © Frank Stepanski

Lessons, files and content of these classes cannot be reproduced and/or published without the express written consent of the author.

Use of this site implies agreement with the [Terms of Use](#)