

Unobtrusive JavaScript Programming

Instructor: Frank Stepanski

Overview

From this class, you will learn more in depth features of the JavaScript language to create some helpful interactive scripts that will work with your existing web pages. We will learn about the three layers of web development, how unobtrusive web scripting works, how to create reusable objects, understand how to use properties and methods of the DOM, debugging scripts with [Firebug](#), cool features of JavaScript libraries like [jQuery](#) and a brief overview of Ajax.

Audience for this Class

I will assume you have either taken the [JavaScript Modification](#) class, which teaches students the basics of JavaScript, or you already have a good understanding of JavaScript already. You also should have a good understanding of web design with HTML and CSS.

This class can be considered an “intermediate” level class because I will be going straight into concepts without reviewing basic syntax of the language.

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](#).

Unobtrusive Scripting

In 2002, [Stuart Langridge](#) coined the term “unobtrusive scripting” which represented the first serious attempt to embed JavaScript in the new theory of CSS-based, standards-compliant web sites.

An unobtrusive script should have all of the following traits:

- It should be usable
- It should be accessible (i.e. if JavaScript doesn't work, the page should remain readable and understandable)
- It should be easy to implement; typically a web developer only has to include the script itself and some JavaScript hooks in the document, and the script works. (We'll discuss JavaScript hooks a little later)
- It should be separate; it resides in its own .js file instead of being scattered though the HTML.

The Three Layers

A web page consists of three layers:

1. HTML structure
2. CSS Presentation
3. JavaScript behavior

The HTML structure layer is the most basic part of the page. The HTML tags form the structure of the page and give meaning to its content. For instance, if a certain text is marked up with an `<h1>`, it's a header and should be treated as such. Browsers are expected to distinguish this header from the surrounding normal text, for example, by displaying it in bold and in a larger font. Once you've created structurally correct HTML, you can be reasonably certain that most browsers will recognize it as a header.

The purpose of the CSS presentation layer is to define how your HTML should be presented. CSS allows you to specify colors, fonts, and other typographical elements, as well as the general layout of the page.

The JavaScript behavior layer adds interactivity to an HTML/CSS page. As soon as you want something to happen when the user mouse's over an HTML element, you need JavaScript to implement the effect.

Every web page needs an HTML structure layer --- without HTML, there is no web page. However, the CSS and JavaScript layers are optional. Old, obscure, or unusual

browsers may not fully support CSS and/or JavaScript, in which case one or both layers may go missing --- the presentation or behavior instructions are never executed.

Separation of Concerns

Another point raised by the division of client-side code into three layers is the separation of concerns. In general, it's best to manage each of the three layers separately. At the most basic level, this is done by making sure of the following:

- The HTML is structured, not too complex and makes sense without CSS and JavaScript
- The CSS presentation layer and the JavaScript layer reside in separate .css and .js files.

Separating concerns makes it easy maintenance. When you separate CSS and JavaScript files, it's easy to link these files from all pages in your site.

CSS Modification

JavaScript allows you to modify CSS, for example, you can make a link red in CSS, and later overrule this style with JavaScript and make the link green. This can be quite useful since style changes allow you to pull the user's eyes to the HTML element you need to focus on --- an error message, for instance.

Hooks

Many scripts don't work on an entire HTML page, but on specific HTML elements within the page. In order to tell a script you created which HTML elements it should work on, you need hooks. A hook can be any kind of HTML structure, although an attribute, such as *id*, is the most obvious choice.

When the script starts up, it goes through the entire document in search of these hooks, and when it finds one, it initializes the script for the HTML elements that contains the hook.

ID

The best known and most popular hook is *id*.

```
var x = document.getElementById('hook');
```

```
// initialize x
```

The script initializes the behavior on the element with *id="hook"*;

This is the simplest way of creating hooks, because the *getElementById* method does all the dirty work for you. Unfortunately, you can only use a certain id once per document.

Note: We will be jumping into the DOM a lot in the next lesson.

Opening a New Window

Let's use the example of opening a new window for a hyperlink as our first example of unobtrusive scripting:

```
<a href = "javascript:popUp('http://www.espn.com/');">A bad link</a>
```

This script is not unobtrusive because it's embedding JavaScript directly in the HTML structure by calling the function *popUp* within the HTML structure layer and using the non-standard pseudo-protocol in of *javascript*. If JavaScript was unavailable or there was an error in the function itself, the hyperlink would not work.

A variation of this would be:

```
<a href = "#" onclick="popUp('http://www.espn.com/'); return false;">A bad link</a>
```

This script is writing JavaScript within an event handler *onclick*, which makes it not unobtrusive and would not work if JavaScript were unavailable.

A slightly better version that would still work if JavaScript were unavailable would be:

```
<a href = "http://www.espn.com" onclick="popUp('http://www.espn.com/'); return false;">A slightly better link</a>
```

You are still writing JavaScript in the HTML structure layer, but it would still work if JavaScript were unavailable because you're putting a valid url link in the *href* attribute value. Having the functionality of the web page still function even if your script is not available is said to have graceful degradation.

Another variation on this last example would be this:

```
<a href = "http://www.espn.com" onclick="popUp(this.href); return false;">A slightly better link</a>
```

Same issue as before but requires a little less maintainability of only entering the url link in once.

The best solution would be this:

```
<a id = "popup" href = "http://www.espn.com/">An unobtrusive link</a></p>
```

This would be unobtrusive because there is no JavaScript embedded in the HTML structure layer. The **hook** would be *id = popup* which would contain all the code in a separate .js file.

This code would find the HTML element with that id and execute the function code that would put the new url in a new window.

Something like this (Figure 1):

```

1  function popUp(winURL) {
2
3      window.open(winURL, "popup", "width=800,height=600");
4  }
5
6  function prepareLink(id) {
7
8      if (!document.getElementById) return false; // Object detection
9
10     var link = document.getElementById("popup");
11     link.onclick = function() { // anonymous function
12
13         popUp(this.getAttribute("href"));
14         return false;
15     }
16 }

```

Figure 1 – Functions that will grab the element and create new window

The *prepareLink (id)* function first checks that the browser supports the *getElementById* function (browser detection) and assuming it is a modern browser, it grabs the element that has an *id* of *popup*. It then assigns the *popup(winURL)* function to the *onclick* event of the element and cancels the default behavior.

Note: An anonymous function is a very common coding technique in JavaScript for assigning a block of code to an event without creating a separate function for it.

The *popup (winURL)* function just opens a new window at a specific size. This is a standard script that you can find on most resource web sites.

The last piece of code will be to show how this will get executed when the page has finished loading.

```

18 window.onload = function () {
19
20     prepareLink("popup");
21 }

```

Figure 2 – Function to execute when page finishes loading

To execute our *prepareLink(id)* function after the page loads, we need to assign it to the *window.load* event. This will execute any number of functions after the document has finished loading.

Note: Creating an anonymous function is an easy way to assign an event multiple functions to execute like *window.load*. Almost every script you will write for this class will use it so make sure you understand it. It's pretty straightforward, but any questions at all just ask. 😊

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml">
4 <title>Unobtrusive JavaScript - Lesson 1</title>
5 <head>
6 <script src="popUp.js" type="text/javascript"></script>
7 </head>
8 <body>
9
10 <a id = "popup" href = "http://www.espn.com/">An unobtrusive link</a>
11
12 </body>
13 </html>
```

Figure 3 – No JavaScript is embedded in document other than to link to external .js

Note: I will be further expanding this script in a later lesson to accommodate multiple links. Don't worry. 😊

The main reason behind unobtrusive JavaScript would be:

“We should only use JavaScript to enhance a functionality that is already given, and not rely on it. JavaScript can be turned off or filtered out by proxies and firewalls of security-aware companies. We can never take it for granted.”

Now that you got a little taste of how unobtrusive JavaScript works, I'm sure you will want to learn more. In the upcoming lessons, I will continue to dissect some popular uses

of JavaScript and create scripts that are unobtrusive, maintainable and object-based to help you further your JavaScript programming skills.

Browser Compatibility

Years ago when JavaScript and the dreaded “DHTML” craze came onto the scene there was specific browser testing that had to be done between the two most popular browsers: IE and Netscape. This was necessary because of differences in how each browser handled the specific DOM functions. Fast forwarding to current day, the two [most popular browsers](#) being IE and Firefox (combined holding around a 93% share) have adopted the majority of the [ECMA-262 standard](#) so the JavaScript that you develop “should” work seamlessly.

For this class I will only be testing the example scripts I include in my weekly .zip files to work in those two browsers (sorry Safari users).

Note: There are no significant differences in JavaScript compatibility between IE 7 and IE 6. The main differences between the two versions (besides the bells and whistles) are in its CSS implementation.

Check out this document to learn all the main differences between the IE browsers:

<http://msdn.microsoft.com/en-us/library/aa155113.aspx>

Resources

<http://www.sitepoint.com/article/unobtrusive-javascript>

<http://www.sitepoint.com/article/javascript-from-scratch>

Assignment for Lesson 1

1. Review the lesson and the example files in wk1.zip.
2. Try to find a script(s) (or one you have already used) and identify where it is not unobtrusive. If it is a long script you just need to include the snippet of code that would need to be changed to make it unobtrusive. You can include a little description of what the script does as well.
3. Post it on the class discussion board.

I will respond with a review on the class discussion board.

Copyrighted 2008 © Frank Stepanski

Used with Permission :: LVS Online Classes / LVS Associates

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](#)