JavaScript Conventions

The following conventions are meant to make our JavaScript code cleaner, more consistent, and less prone to errors. As our users continue to expect a more interactive and real-time experience, JavaScript becomes increasingly important. As powerful as JavaScript is, it lacks the structure, type safety, and most of the IDE features we are used to with C#. Because of this, the asynchronous nature of JavaScript, and the browser environment it runs in, things tend to get complex and messy very quickly. JavaScript code is more prone to logic errors, memory leaks, compatibility issues, turning into *spaghetti code*, and noticeable performance issues (caused by inefficient code).

We will all be expected to follow these conventions going forward. The only exception is with legacy code. If using these conventions within existing code makes it confusing to follow (because of using two different styles of coding) then use the existing coding style, or rewrite the existing code. There are changes you need to make in Visual Studio, so make sure you install any extensions mentioned and make any settings changes indicated.

These conventions will continue to change as we learn more about JavaScript and encounter different issues. Suggestions for improvements are encouraged. You can find the most up-to-date copy of this file on **P:\Dev\Mitch\js**.

1. Use JSHint for validation.
    a. JSHint, which is built on top of JSLint with more options, is a tool used to check if your JavaScript code meets certain coding standards.
    b. Requires a Visual Studio extension
        i. **Visual Studio 2010**: Install the **JSLint.VS2010** visual studio extension (**Tools > Extension Manager**). After installing do the following:
            1) Close and reopen visual studio.
            2) Go to **Tools > JS Lint Options** and click **Import**. Choose the settings file located at **P:\Dev\Mitch\js\JSLintOptions.xml**
        ii. **Visual Studio 2012/2013:** Install the **Web Essentials** extension if it isn't already installed. This comes with many tools, including JSHint validation. After installing do the following:
            1) Open a project or solution in visual studio, go to **Web Essentials > Edit global JSHint settings**, and replace the contents with what's in **P:\Dev\Mitch\js\.jshintrc**
            2) Go to **Tools > Options > Web Essentials > JavaScript**, and under **Linter**, set **Results location** to **Error**, **Run on build** to **False**, and **Run on save** to **True**.
            3) You need to disable JSCS, which is another JavaScript validator that will conflict with JSHint. Go to **Web Essentials > Edit global JSCS settings** and replace the contents with the following:

               { "excludeFiles": [/] }**
        iii. There will be an up-to-date copy of the JSHint settings for both extensions on **P:\Dev\Mitch\js** so you might want to update these settings periodically.
    c. **NOTE:** Not all coding conventions are listed in this document. Some conventions JSHint will warn you about, and a web search for the error will give you more details.

2. Strict Mode
    a. It's good practice to always use strict mode. It helps get rid of logic errors by tightening the rules you must follow to produce valid JavaScript. If you aren't careful in normal mode, it's easy to write code that is technically valid but doesn't work as expected.
    b. Put **'use strict';** as the first statement in the function that contains all of the code in a file.
    c. **'use strict';** must be contained within a function, because if it's in the global namespace it will conflict with other files. Third-party libraries and other files may not have been written for strict mode.

```
(function () {
  'use strict';

  // code goes here

}());
```

3. All code should be isolated in its own scope. You should not be polluting the global scope. The best way to achieve this is to wrap all of the code in a file within a self-invoking anonymous function, and then only expose what needs to be accessed outside of the file through a namespace. That namespace (object) should be the only thing added to the global scope.

    a. Self-invoking anonymous function:

```
(function () {
  'use strict';

  // code goes here

}());
```

    b. Create a namespace (global object) by using a self-invoking anonymous function:

```
var GlobalNamespace = (function () {
  'use strict';

  var privateVariable = 'This is private',
      globalVariable = 'This can be accessed in the global namespace';

  return {
    GlobalVariable: globalVariable
  };

}());

console.log(GlobalNamespace.GlobalVariable);
```

c. You can pass in global variables to the new scope you created so the scope has a local copy. This has several benefits:
    i. When accessing a local variable as opposed to a global variable, it doesn't have to go up the scope chain to find it.
    ii. Your code becomes more modular and loosely coupled since you aren't accessing the global variable directly. If the name of the global variable changes, you only have to change it in one place (where it's passed in) and not everywhere within your code that uses it.
    iii. It's an easy way to create a shorter variable name.

```
(function ($) {
  'use strict';

  // This has all the benefits listed, but also
  // it lets you use the shortcut variable $ for jQuery
  // without worrying about possible conflicts with other
  // libraries.
  $(document).ready(function () {
    ...
  });

}(jQuery));
```

4. Variable declarations
    a. Declare all variables at the top of the function they are contained in. In JavaScript, the only code block that creates a new scope is a function, and because of variable hoisting, no matter where you declare a variable inside a function, it is "hoisted" to the top when it is parsed anyway. Doing this prevents declaring a variable more than once and it makes the code read exactly how it is parsed.
    b. Only use one **var** keyword in a function and separate variable declarations with commas.
    c. Only one variable declaration per line.
    d. Declare constants first.
    e. For primitive types, always give the variable a default value rather than leave it uninitialized.
        i. This helps clarify what type the variable is since JavaScript isn't strongly typed.
    f. Variables containing functions or other complex types can be left uninitialized when declared.
        i. These variables should be declared last.

```
var MAX_ELEMENTS = 10,
    description = '',
    isValid = false,
    numElements = 0,
    incrementElements;

incrementElements = function () {
  if (numElements < MAX_ELEMENTS) {
    numElements++;
  }
};
```

5. Variable naming convention
    a. Namespace, class, and object/class property names start with an uppercase letter (*PascalCase*).
    b. Local variable names are *camelCase*.
    c. Constants have names with all caps and underscores separating words (*CONSTANT_NAME*).
        i. The only exception to this rule is server-side enums, which when represented in JavaScript, will be using *PascalCase* to match the server-side naming convention.
    d. Names of variables containing a jQuery collection of DOM elements should begin with **$** and be camelCase.

```
var localVariable = function () {
  return {
    CONSTANT_VALUE: 10,
    ObjectProperty: 'Foo bar',
    $inputElements: $('input')
  };
};
```

6. jQuery
   a. Always cache jQuery collections of DOM elements if using the collection more than once so that you aren't doing a lookup by selector more than once.

```
// Wrong:
var setSidebar = function () {
  $('.sidebar').hide();

  $('.sidebar').css({ 'background-color': 'pink' });
};

// Right:
var setSidebar = function () {
  var $sidebar = $('.sidebar');

  $sidebar.hide();

  $sidebar.css({ 'background-color': 'pink' });
};
```

7. Code blocks
   a. Put opening brackets **{** on same line (does not create a new line).
   b. There should be a space in between the keyword and the opening parenthesis, a space before and after any binary operators, and a space between the closing parenthesis and the opening bracket.

```
// Wrong:
if (num === 1)
{
  //...
}

// Right:
if (num === 1) {
  //...
}
```

   c. All code blocks should be wrapped in brackets **{}**. Even though certain types of statements (**if**, **for**, etc.) allow you to omit the brackets if the body of the statement is only one line, still add the brackets. This makes it more readable, and there is less chance of error when adding more lines inside the statement in the future.
   d. To condense the code, put the entire statement on one single line with brackets, rather than two lines with no brackets.

```
// Wrong:
if (num === 1)
  num++;

// Right:
if (num === 1) {
  num++;
}

// or:

if (num === 1) { num++; }
```

8. Use single quotes for consistency, and because it helps when escaping a block of html, which normally uses double quotes.

9. Always use **===** and **!==** rather than **==** and **!=.** This will ensure that it is doing an exact match based on value and variable type.

10. Max line length of 90 characters. JSHint will validate this for you.

11. Use 2 spaces for tabs
    a. In Visual Studio, goto **Tools > Options > Text Editor > JScript > Tabs** and enter **2** for both **Tab Size** and **Indent Size** and make sure **Insert Spaces** is selected.

12. Line breaking
    a. When breaking on a comma, put the comma at the end of the first line as opposed to the beginning of the new line.
    b. When breaking on an operator, put the operator at the beginning of the new line

```
var descriptions = [
  'This is a one-line description.',
  'This is a description broken into '
  + 'two lines.'
];
```

13. Define all functions as an anonymous function assigned to a variable (function expression) rather than using the **function** keyword followed by a function name (function declaration).
    a. This is for consistency, and by treating a function just as another variable type, it behaves the same way as any other variable and we don't need to worry about how function declarations differ from variables. We don't lose any functionality by doing this.
    b. When defining an anonymous function, there should be a space between the **function** keyword and the first parenthesis. This distinguishes it as a function definition rather than invoking/calling a function.

```
// Wrong:
function myFunction() {
  //...
}

// Right:
var myFunction = function () {
  //...
};
```

14. Use factory pattern over using classes.
    a. Avoid using constructors and the **new** keyword to instantiate instances of a class like class-based languages. This is a built in nicety that attempts to make code for JavaScript's prototypal inheritance model familiar and similar to the classic OO class-based model. In reality, the prototypal model works differently and there are nuances that are hidden when using "classes" in JavaScript. Instead you should use the factory pattern, which is much more flexible. In the most basic form, this just means you write a function that creates an object for you rather than *newing* up an instance of a class.
    b. Also, avoid using the **new** keyword at all.

```
// Wrong:
var myArray = new Array();

// Right:
var myArray = [];
```