JavaScript Coding Standards

Ajax Control Toolkit

Monday, 10 May 2010

Version 1.0

Prepared by

Simon Ince, Stuart Leeks

Contributors

Dave Reed, Bertrand Le Roy, Stephen Walther

Revision and Signoff Sheet

Change Record

Date	Authors	Version	Change reference
September 2009	Simon Ince Stuart Leeks	0.1 – 0.6	Draft
October 2009	Simon Ince Stuart Leeks	0.7	Changes applied following review by Matthew Farmer Released for internal review
November 2009	Simon Ince Stuart Leeks	0.8	Applied feedback from reviewers.
January 2010	Simon Ince Stuart Leeks	0.9	Incorporated feedback from reviewers and altered progressive enhancement and accessibility guidelines.
March 2010	Simon Ince Stuart Leeks	0.95	Updated to reflect product announcements. Released for Insiders review.
April 2010	Simon Ince Stuart Leeks	1.0	Updated following Insiders review comments.

Reviewers

Name	Version reviewed	Position	Date
Stephen Walther	0.8	Product group	January 2010
Bertrand Le Roy	0.7	Product group	September 2009
Dave Reed	0.7	Product group	September 2009
Matthew Farmer	0.6	UK MCS	September 2009
ASP.NET Insiders	0.95	Various	March 2010

Table of Contents

1 Introduction & Scope

These standards are intended to be used for software written in JavaScript using the Ajax Control Toolkit.

They have been designed to maintain consistency with the published "Design Guidelines for Developing Class Libraries" and internal Microsoft coding standards for the .NET Framework where possible, whilst prioritizing JavaScript compatibility, performance, and maintainability.

The content is split into sections that can be consumed relatively independently, to enable you to get started quickly;

- "Naming Standards" is simply a set of conventions for naming JavaScript language elements.
- "Style Guidelines" are preferred ways to structure and format JavaScript code.
- "Design Guidelines" define recommended practices to create maintainable code.
- "Programming Guidelines" specifies practices that are beneficial to the JavaScript environment, from a performance, reliability, maintainability, and compatibility point of view.

Further guidance is available from the Microsoft ASP.NET and Ajax product group blogs, from Patterns and Practices [1], and on the ASP.NET Community Site [2].

The standards in this document do not cover all topics that must be considered when building applications with JavaScript, and therefore it is recommended that other information sources are consulted in partnership. In particular, consider seeking guidance on;

- JavaScript Security, such as Cross Site Scripting and Request Forgery mitigations.
- Accessibility and browsing device support
- Usability and User Interface design

[1] Patterns & Practices Web Application Guidance: http://webclientguidance.codeplex.com/

[2] ASP.NET Community Site: http://www.asp.net/

2 NAMING STANDARDS

This section presents standards for naming JavaScript language elements. Naming consistency is an essential first step to improving the clarity, readability, and therefore maintainability of code when many developers are working on it.

The standards in this section have been adapted to JavaScript from those defined for other languages, in particular those that have well established and proven naming standards. The adaptation process has considered and prioritized JavaScript compatibility and performance.

2.1 Capitalization Styles

2.1.1 Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:

```
BackColor
```

2.1.2 Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

```
backColor
```

2.1.3 Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

Wrong:	`Io`	Wrong:	`TCP`
Right:	`IO`	Right:	`Tcp`

2.1.4 Hungarian Notation

The identifier is preceded by text that indicates the data type it represents. For example, a string variable may be named `strValue`, and an integer `intValue`.

This approach is not recommended for any JavaScript language element.

2.2 Capitalization and Case Sensitivity

- Do not create two namespaces with names that differ only by case.
- Do not create a namespace with type names that differ only by case.
- Do not create a function with parameter names that differ only by case.
- Do not create a type with property, method, field or event names that differ only by case.

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	Behavior
Enum type	Pascal	MouseButton
Enum values	Camel	leftButton
Event	Camel	load
Exception class	Camel	Error.argument
Static field	Camel	redValue
Interface	Pascal	IDisposable Note Always begins with the prefix `I`.
Method	Camel	toString
Namespace	Pascal	Sys.Net
Parameter	Camel	typeName
Property	Camel	backColor
Local variable	Camel	var myVariable
Public instance field	Camel	horizontal

2.3 Abbreviations

To avoid confusion follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use getWindow instead of getWin.
- Do not use acronyms that are not generally accepted in the computing field or your particular business context.

- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use UI for User Interface and Olap for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use HtmlButton or htmlButton. However, you should capitalize acronyms that consist of only two characters, such as Sys.UI instead of Sys.Ui.
  Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use camel case for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

2.4 Conventions

2.4.1 Type visibility

JavaScript has no concept of public or private members other than that determined as a result of variable scope. Therefore, when writing Object Oriented JavaScript, prefix members that are intended to be private with an underscore ("_"). For example, in Sys.EventHandlerList, a private member named "_list" is declared:

```
Sys.EventHandlerList = function EventHandlerList() {
    this._list = {};
}
```

Visual Studio also excludes members prefixed with an underscore from Intellisense.

2.4.2 Uniqueness

Avoid using the same name for multiple type members, such as events and properties. Ensure each is unique to maintain clarity of code.

2.5 Namespaces

The general rule for naming namespaces is to use the company name followed by the technology name. This is to avoid the possibility of two published namespaces having the same name.

```
CompanyName.TechnologyName[.Feature]
```

For example:

```
Microsoft.Media
Microsoft.Office.Word
```

Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies.

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the Sys.UI depend on the classes in Sys. However, the classes in Sys do not depend on the classes in Sys.UI.

You should use Pascal case for namespaces, and separate logical components with periods, as in `Microsoft.Office.PowerPoint`. If your brand employs non-traditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the fictional namespaces `Microsoft.newProducteXample` and `contoso.secureIT` illustrate appropriate deviations from the Pascal case rule.

Use plural namespace names if it is semantically appropriate. For example, use `Sys.Collections` rather than `Sys.Collection`. Exceptions to this rule are brand names and abbreviations.

Do not use the same name for a namespace and a class. For example, do not provide both a `Debug` namespace and a `Debug` class. Namespaces are referred to regularly in script and therefore should be kept appropriately brief to minimize download size whilst still being descriptive. For example, the Ajax Control Toolkit uses `Sys` instead of `System`. Similarly, avoid nesting namespaces too deep to keep them succinct, such as `Sys.UI` rather than `Sys.UI.Controls`.

Finally, see section 3.7.2 for guidance on naming JavaScript files.

2.6 Type naming

2.6.1 Classes

- Use a noun or noun phrase to name a class.
- Use Pascal case.
- Use abbreviations sparingly.
- Do not use a type prefix, such as `C` for class, on a class name. For example, use the class name `Dialog` rather than `CDialog`.
- Do not use the underscore or dollar characters (_ and $).
- Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name `IdentityStore` is appropriate, as is the interface name `IIdentityStore`.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, `ArgumentException` is an appropriate name for a class derived from a class named `Exception`, because `ArgumentException` is a kind of `Exception`. Use reasonable judgment in applying this rule. For example, `CustomButton` is an appropriate name for a class derived from `Control`. Although a customized button is a kind of control, making `Control` a part of the class name would lengthen the name unnecessarily.

2.6.2 Interfaces

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name `IContainer` uses a descriptive noun. The interface name `ICustomAttributeProvider` uses a noun phrase. The name `IDisposable` uses an adjective.
- Use Pascal case.
- Use abbreviations sparingly.
- Prefix interface names with the letter `I`, to indicate that the type is an interface.

- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter `I` prefix on the interface name.
- Do not use the underscore or the dollar character (_ and $).

2.6.3 Enumerations

For guidance on defining enumerations, see section 4.3.6.

- Use Pascal case for `Enum` types and camel case for value names.
- Use abbreviations sparingly.
- Do not use an `Enum` suffix on `Enum` type names.
- Use a singular name for `Enum` types.

2.7 Type members

2.7.1 Functions

- Use verbs or verb phrases to name methods.
- Use camel case for method names.

Use an underscore ("_") prefix for methods that are intended to be private (see 2.4.1). To ensure that JavaScript debuggers and profilers can correctly infer a type member's name, avoid anonymous functions in debug scripts. For example:

Wrong: `Sys.toString = function () { ... code ...}`
Right: `Sys.toString = function toString() { ... code ...}`

Release versions of these scripts usually omit function names to reduce the size of the download.

For information on ensuring that these function names are kept unique and do not clash with other scripts see section 3.7.1.

2.7.2 Properties

- Use a noun or noun phrase to name properties.
- Use camel case (get_myProperty, set_myProperty).
- Do not use Hungarian notation (see section 2.1.4).
- If possible create a property with the same name as its underlying type. For example, if you declare a property named "color", the type of the property should likewise be Color.

Property accessors should always be prefixed with "get_" and "set_" respectively. For example:

```
MyApp.Person.get_name =
    function get_name() { return _name; }
```

2.7.3 Fields

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.

- Use camel case.
- Do not use a Hungarian notation prefix on static field names.
- Use an underscore ("_") prefix for fields that are intended to be private (see 2.4.1)

```
this._name = '';
```

2.7.4 Events

- Use camel case (add_myEvent, remove_myEvent).
- Do not use Hungarian notation.
- Specify two parameters named `sender` and `args`. The `sender` parameter represents the object that raised the event. The state associated with the event is encapsulated in an instance of an event class named `args`. Use an appropriate and specific event class for the `args` parameter type.
- Name an event argument class with the `EventArgs` suffix.
- Consider naming events with a verb. For example, correctly named event names include `clicked`, `painting`, and `droppedDown`.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a `close` event that can be canceled should have a `closing` event and a `closed` event. Do not use the `beforeXxx`/`afterXxx` naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use `close` instead of `onClose`.

Event accessors should always be prefixed with "add_" and "remove_" respectively. For example:

```
function add_click() { ... code ...}
function remove_click() { ... code ...}
```

For further information on defining events see section 4.4.7.

2.8 Parameters

- Use camel case for parameter names.

```
Wrong:   function doSomething(InputData)
Right:   function doSomething(inputData)
```

- Use descriptive parameter names in Debug script versions. Release scripts may use abbreviated names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.

```
Wrong:   function copy(a, b) { ... code ...}
Right:   function copy(source, destination) { ... code ...}
```

- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter‘s type if Documentation Comments are used (see section 3.4.1). Therefore, a parameter's name can be put to better use by describing meaning.

Wrong: `function doSomething(str, integer) { ... code ...}`
Right: `function doSomething(message, length) { ... code ...}`

- Do not prefix parameter names with Hungarian type notation (see section 2.1.4).

Wrong: `function doSomething(strMessage, intLength) { ... code ...}`
Right: `function doSomething(message, length) { ... code ...}`

3 STYLE GUIDELINES

Style Guidelines are preferred ways to structure and format JavaScript code to further enhance readability and consistency across development teams. They may be considered optional by some development teams.Where possible the guidelines have been designed to complement developer tooling, and vice-versa.

3.1 Tabs and indenting

Tab characters (\0x09) should not be used in code. All indentation should be done with 4 space characters.

3.2 Bracing

Open braces should always be at the end of the line with the statement that begins the block. Contents of the brace should be indented by 4 spaces. For example:

```
if (someExpression) {
    doSomething();
}
else {
    doSomethingElse();
}
```

This style helps to eliminate occurrences of the semi-colon bug; that is, by design JavaScript will automatically insert a semi-colon after some statements. Adding the opening brace on the same line ensures the parser understands that the statement is intended to be split over multiple lines.

When using a “switch” statement, “case” statements should be indented as follows;

```
switch (someExpression)  {
    case 0:
        doSomething();
        break;
    case 1:
        doSomethingElse();
        break;
}
```

Braces should never be considered optional in a Debug script file. Even for single statement blocks, you should always use braces. This increases code readability and maintainability.

```
for (var i=0; i<100; i++) { doSomething(i); }
```

Minifiers may choose to remove these braces for a Release script to reduce the download size. This is best done using an automated tool, allowing the Debug script to adhere to the standard (see section 3.7.3).

3.3 Quotes

As a general rule, use single quotes to delimit strings. This means HTML fragments in code can contain double quotes inside literal strings with ease. For example;

```
var content = '<span id="myspan">…';
```

Note, however, that it is preferable to avoid emitting mark-up from code as this can complicate maintenance for layout changes.

3.4 Commenting

Comments should be used to enhance the reader's understanding of the code with respect to its intention, algorithmic overview, and/or logical flow. Comments that do not enhance this understanding should be omitted. While there are no minimum comment requirements and some routines need no commenting at all, it is a good practise to include comments reflecting the programmer's intent and approach.

3.4.1 Documentation comments

All methods intended to be consumed by other classes (i.e. public members, see section 2.4.1) should use XML documentation comments in debug script versions. Release versions should exclude comments.

```
function doSomething(index) {
    /// <summary>
    /// Performs an action on the data item indicated by index.
    /// </summary>
    /// <param name="index">An index into the list of data</param>
    /// <returns>true for success, false for failure</returns>

    return this._action(_data [index]);
}
```

Consider using AjaxDoc [3] to extract these comments and generate documentation.

3.4.2 Comment Style

The // (two slashes) style of comment tags should be preferred over the /* */ syntax in most situations. Where possible, comments should be placed above the code that they pertain to rather than beside it.

3.5 Spacing

Good use of spaces improve readability by decreasing code density, and are removed by script minifier tools (see section 3.7.3) so should not affect download size. Here are some guidelines for the use of space characters within code:

[3] AjaxDoc: http://www.codeplex.com/ajaxdoc

Do use a single space after a comma between function arguments.

Wrong: `doSomething(myChar,0,1);`

Right: `doSomething(myChar, 0, 1);`

Do not use a space after the parenthesis and function arguments.

Wrong: `doSomething ( myChar, 0, 1 )`

Right: `doSomething(myChar, 0, 1)`

Do not use spaces between a function name and parenthesis.

Wrong: `doSomething ()`

Right: `doSomething()`

Do not use spaces inside brackets.

Wrong: `x = dataArray[ index ];`

Right: `x = dataArray[index];`

Do use a single space before flow control statements.

Wrong: `while(x===y)`

Right: `while (x === y)`

Do use a single space before and after comparison operators.

Wrong: `if (x===y)`

Right: `if (x === y)`

3.6 Object Constructors

Prefer using JSON syntax to create new empty arrays or objects.

Wrong: `this._dataItems = new Array();`

Right: `this._dataItems = [];`

Wrong: `this._dataItem = new Object();`

Right: `this._dataItem = {};`

In general avoid using built in object constructors (such as Number) as this results in a boxed object, which can be costly in performance.

3.7 File Organization

3.7.1 Impact of Structure on Scope

Consider wrapping sections of script in a closure to guarantee that functions are uniquely named. This eliminates the previous practice of creating long function names, including dollar signs to separate sections of the name. These function names could become truncated by tooling and therefore of limited use.

For example, the following class declaration uses a long function name to identify the "tick" method:

```
MyNamespace.MyClass = function MyClass () {
    this._interval = 1000;
    this._enabled;
    this._timer = null;
}

MyNamespace.MyClass.prototype = {
    _tick: function MyNamespace$MyClass$tick() {
        alert('Ticked...');
    }
}

MyNamespace.MyClass.registerClass('MyNamespace.MyClass');
```

Instead, wrap this script in a closure as follows:

```
(function(window) {

    MyNamespace.MyClass = function MyClass () {
        this._interval = 1000;
        this._enabled;
        this._timer = null;
    }

    MyNamespace.MyClass.prototype = {
        _tick: function tick() {
            alert('Ticked...');
        }
    }

    MyNamespace.MyClass.registerClass('MyNamespace.MyClass');

}) (window);
```

The "tick" method no longer requires a long name to ensure that it is unique, yet it will be correctly reported in debug and profiling tools.

The remainder of this document assumes this approach has been used to permit short function names.

3.7.2 Naming and Structure

Source files should generally contain whole related areas of functionality, not just a single public type. This does increase the likelihood of additional source code control conflicts and merges when multiple developers are contributing, but minimizes the number of script downloads or script combines required in the application, and ensures script references are kept brief and clear.

- Source files should ideally match the root namespace for the contents, or be the commonly accepted name of the framework they contain
- Consider embedding a file version in the filename to improve cache behavior

- Include the ".debug" indicator for debug scripts.
- Do not include any equivalent indicator for release scripts.

For example, "App.UI-1.0.1.js" would contain version 1.01 of the release build for the App.UI namespace. "App.UI-1.0.1.debug.js" would contain the equivalent debug script.

Script imports using the <reference> notation should appear at the top of the file. Comments describing the contents of the file should follow, and finally the JavaScript itself.

3.7.3 Minification

Release scripts should be processed using a script minifier, to reduce the size of the download and optionally introduce some obfuscation. One such minifier is the Microsoft Ajax Minifier [4].

3.7.4 NotifyScriptLoaded

The call to `Sys.Application.notifyScriptLoaded` is deprecated, and should therefore be removed from any files that target the current toolkit version.

3.7.5 Script Loading

Ensure that JavaScript files are configured to be cacheable by the browser when possible, to optimize the responsiveness of pages that reference them.

The Script Loader included with the Ajax Control Toolkit is the preferred way to manage the downloading of scripts and their dependencies. It can download multiple scripts in parallel and ensure that prerequisite scripts are loaded in the right order.

For guidance on using the Script Loader for custom scripts and on tuning JavaScript file downloads see the patterns & practices Web Client Guidance [5].

[4] Microsoft Ajax Minifier: http://www.asp.net/ajaxLibrary/download.ashx

[5] p&p Web Application Guidance: http://webclientguidance.codeplex.com/

4 Design Guidelines

The design guidelines in this section are intended to encourage good practices that lead to maintainable, supportable, well performing script.

4.1 Design Approach

Two common approaches to JavaScript development exist, sometimes referred to as "static" and "dynamic". A static approach consists of defining script components as encapsulated JavaScript classes and their associated functions, organized into namespaces. A dynamic approach leans more towards creating small highly specialized functions, often combined to have the complete desired effect.

Both of these approaches are completely valid, and this document does not aim to prefer one over the other. However, a static approach is likely to contain more code elements (such as namespaces and classes), and therefore more of this document's content is applicable.

4.2 Progressive Enhancement

When building a Web Site consider using *Progressive Enhancement*. This results in a page that is fully functional without script; the functionality and user experience are enhanced when script is enabled. This is particularly well complimented by the principles of Unobtrusive JavaScript [6]. Web pages created in this manner do not have a mandatory dependency on JavaScript, and therefore are more likely to be compatible with a wider range of clients and accessibility tools, and to be considered Search Engine Optimized (SEO).

However, requiring no dependency on JavaScript can be unrealistic for some applications, and adds significant development effort. Developers should therefore weigh the requirements for each project against the effort involved in following this practice. For further guidance on Progressive Enhancement refer to the Web Client Guidance project from patterns & practices [7].

4.3 Types

4.3.1 Namespace

Do use namespaces to organize types into a hierarchy of related feature areas.

Avoid deep namespace hierarchies. Such hierarchies can make it difficult for a developer to locate the functionality they require, and lead to unnecessary increased script size and hence page weight.

Avoid empty namespaces or extending a namespace without adding value. Avoid having too many namespaces.

Types that are used in the same scenarios should be in the same namespaces when possible.

---

[6] Unobtrusive JavaScript: http://en.wikipedia.org/wiki/Unobtrusive_Javascript

[7] p&p Web Application Guidance: http://webclientguidance.codeplex.com/

Do not define types without specifying their namespaces. Types that are not assigned a namespace are placed in the global namespace. This can cause name collisions with other scripts or frameworks, therefore preventing some libraries from being used concurrently on the same page.

4.3.2 Component, Behavior and Control

Select the correct base class for script classes. Use the Type.registerClass method to define the base class for a script when inheritance functionality is required, such as implementing an interface, overriding base class members, or creating Components, Controls or Behaviors:

```
MyNamespace.MyClass.registerClass('MyNamespace.MyClass',
    Sys.IComponent);
```

Use `Sys.Component` as the base class for script that does not have a visual element, but may manage references between other components or DOM elements. Use events and the dispose method to ensure that relationships are managed correctly and released in a timely fashion.

Use `Sys.UI.Behavior` as the base class for script that enhances an existing DOM element. When using ASP.NET Web Forms, associate the script Behavior with a server-side Extender[8]. Behaviors are ideal for progressively enhancing a web page.

Use `Sys.UI.Control` as the base class for script that fundamentally changes a single DOM element's behavior to provide new functionality. When using ASP.NET Web Forms, associate the script Control with a server control [9].

When selecting a base class, prefer `Sys.Component` where possible. If the control has a visual representation through a relationship with a DOM element, prefer `Sys.UI.Behavior` where possible. This means that multiple Behaviors may be applied to a single DOM element, in contrast to a `Control` which takes exclusive ownership of a single DOM element at a time.

4.3.3 IDisposable

Implement `Sys.IDisposable` for script classes that manage references to DOM elements, other script elements, timers, or more.

```
MyNamespace.MyClass.registerClass('MyNamespace.MyClass',
    null,
    Sys.IDisposable);
```

Prefer inheriting from `Sys.Component` over directly implementing `Sys.IDisposable`. `Component` automatically implements `IDisposable`, but also provides further infrastructure and lifetime management for script elements. When directly implementing `IDisposable` without inheriting from `Component`, ensure that `Sys.Application.registerDisposableObject` is called when instances of the class are created.

---

[8] Extenders: http://msdn.microsoft.com/en-us/library/bb386403.aspx

[9] Script Controls: http://msdn.microsoft.com/en-us/library/bb386450.aspx

When implementing the dispose method, ensure that the final task is to call the dispose method on the base class:

```
    MyNamespace.MyClass.callBaseMethod(this, 'dispose');
```

The dispose method should never cause errors, and must allow for being called multiple times.

Ensure that references are released early, event handlers are cleared for the DOM element the current class refers to, event handlers placed on other DOM elements are removed, and timers are stopped. An example dispose implementation follows:

```
dispose: function dispose() {
    $clearHandlers(this.get_element());
    this._buttonReference.removeHandler('click',
        this._clickHandler);
    window.clearInterval(this._tick);
    MyNamespace.MyClass.callBaseMethod(this, 'dispose');
}
```

Note that this example addresses a Control, which has full ownership of a DOM element and therefore can call `$clearHandlers`. Behaviors should not call this method, and instead should use `$removeHandler`.

Finally, ensure member variables that hold references to DOM elements or browser plug-ins are set to null. You may also consider using the delete keyword. For example, use one of the following:

```
    this._events = null;
OR  delete this._events;
```

Dispose methods should also support being called multiple times without causing negative side effects.

4.3.4 Class Design

Define a class when it encapsulates some behavior or data, providing functionality to other script or DOM elements found on a page.

Prefer using a *prototype* approach over using *closures* to define class members. Brief examples of closure and prototype-based approaches are as follows.

4.3.4.1 Closure Approach

```
MyNamespace.MyClass = function MyClass () {
    this._interval = 1000;
    this._enabled;
    this._timer = null;
    this._tick = function() {
        alert('Ticked...');
    }
}
```

The definition of the "tick" method is local to the constructor of "MyClass". This approach requires more memory and processing when multiple instances of MyClass are created in comparison with a prototype approach, shown in the next section. This can be proven with the following simple script;

```
MyClass = function MyClass() {
    this._tick = function() { alert('tick'); };
}
var a = new MyClass();
var b = new MyClass();

// functions are not the same
alert(a._tick === b._tick);

MyClass2 = function MyClass2() { };
MyClass2.prototype = {
    _tick : function tick() { alert('tick'); }
}
var c = new MyClass2();
var d = new MyClass2();

// functions are the same
alert(c._tick === d._tick);
```

4.3.4.2 Prototype Approach

```
MyNamespace.MyClass = function MyClass () {
    this._interval = 1000;
    this._enabled;
    this._timer = null;
}

MyNamespace.MyClass.prototype = {
    _tick: function tick() {
        alert('Ticked...');
    }
}

MyNamespace.MyClass.registerClass('MyNamespace.MyClass');
```

Ensure that functions are separated by commas to avoid parsing errors. For example:

```
MyNamespace.MyClass.prototype = {
    _tick: function tick() {
        alert('Ticked...');
    },
    _tock: function tock() {
        alert('Tocked...');
    }
}
```

The prototype approach does have a small performance cost in requiring members to be looked up through the prototype chain, but this is seen as a worthwhile trade-off.

4.3.5 Interface Design

It is preferable to create classes rather than interfaces. Classes can have new members added in later versions of a framework without breaking compatibility, but interfaces cannot. Do not add members to an interface that has previously been shipped.

Use interfaces when classes that are otherwise unrelated need to share some common functionality, and when classes that share functionality already have logical base classes.

Avoid using marker interfaces (interfaces with no members). Instead consider using a property or status field.

4.3.6 Enumerations

Enumerations provide sets of constant values that are useful for strongly typing members and improving readability of code.

Use an enumeration to strongly type parameters, properties, and return values that represent sets of values. Favor using an enumeration instead of static constants.

Do not define reserved enumeration values that are intended for future use.

Avoid publicly exposing enumerations with only one value.

Do provide a default for simple enumerations. If possible, name this value `none`. If `none` is not appropriate, use a logical term.

Define enumerations using a prototype approach, and register them using the `registerEnum` method. This is demonstrated by the system `Sys.UI.Key` enumeration:

```
Sys.UI.Key = function Key() {
}

Sys.UI.Key.prototype = {
    backspace: 8,
    tab: 9,
    enter: 13,
    esc: 27,
    space: 32,
    pageUp: 33,
    pageDown: 34,
    end: 35,
    home: 36,
    left: 37,
    up: 38,
    right: 39,
    down: 40,
    del: 127
}

Sys.UI.Key.registerEnum('Sys.UI.Key');
```

4.4 Members

4.4.1 Function overloading

Functions in JavaScript are defined by their name, not by their signature as in some other languages, such as C#. Therefore there is no way provided by the core language to define function overloads.

Instead of providing overloads, consider taking advantage of JavaScript's optional parameters; any parameter can be omitted from a call, for example:

```
MyNameSpace.MyClass.prototype.foo = function foo(p1, p2) {
    // other code removed
    if (p2) {
        // use param2 parameter
    }
    // other code removed
}
```

This demonstrates the use of an “if” statement to determine whether a parameter has been provided. See section 5.5 for more information on undefined and null.

To communicate this behavior to a developer, an XML comment can be used to indicate that a parameter is optional, as follows:

```
/// <param name="p2" type="Type"
    optional="true" mayBeNull="true"></param>
```

It is good practice to permit optional parameters to be supplied as null, especially if they are not the last parameter in the list. When there are a large number of optional parameters consider packaging the parameters in an "options" object to improve code readability. For example;

```
// call function
myobject.applySettings(this, {
    width: 10,
    height: 100,
    color: 'red',
    message: 'Do not click here'
});
```

4.4.2 Overriding base class behavior

When overriding methods defined on a base class, it is usually desirable to call the base class implementation too, for example:

```
    function initialize() {
        Sys.UI.Behavior.callBaseMethod(this, 'initialize');
        var name = this.get_name();
        if (name) this._element[name] = this;
    }
```

The call to Type.callBaseMethod ensures this happens as expected.

4.4.3 Properties vs Methods

Class library designers often must decide between implementing a class member as a property or a method. In general, methods represent actions and properties represent data. Use the following guidelines to help you choose between these options.

Use a property when the member is a logical attribute of a type, such as a “Name” property. Use a property if the value it returns is simply a data item stored within a class, and the property would just provide access to the value.

Use a method when:

- The operation is a conversion, such as Object.toString.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the get accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
- The member is static but returns a value that can be changed.
- The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code.

4.4.4 Properties vs Fields

Properties are used in static languages to encourage a class' interface to remain static, and to adhere to behaviour encapsulation guidelines. In JavaScript the penalties of the additional execution time and script download size can outweigh the benefits of strictly using Properties.

Consider using a field when;

- The declaring Type is only used for passing data between methods (such as an Event Args class)
- There is no behaviour or validation required when the value is changed.
- Change notifications are not required.

4.4.5 Property Design

Use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties – use a method instead.

Provide sensible default values for all properties, ensuring that these defaults result in an efficient design. Use `null` as a default if there is no logical default value. `undefined` should never be the return value of a property.

Avoid throwing exceptions from property getters. Preserve the previous value if a property setter throws an exception.

Do allow properties to be set in any order even if this results in a temporary invalid object state. Indicate this state using a status, or disable behavior until the class is correctly configured.

Classes should raise property-changed events if consumers should be notified when the component's property changes programmatically. The `Sys.Component` class implements the `Sys.INotifyPropertyChange` interface so classes inheriting from `Sys.Component` can implement change notification simply by calling `raisePropertyChanged` from property setters. For example:

```
    set_text: function set_text(value) {
        if (this._text !== value) {
            this._text = value;
            this.raisePropertyChanged('text');
        }
    }
```

4.4.6 Constructor Design

Consider providing simple, ideally default, constructors. A simple constructor has a very small number of parameters, and all parameters are primitive types, enumerations, or a reference to a DOM element.

Use constructor parameters as shortcuts for setting main properties. Consider allowing parameters to be optional as a substitute for constructor overloads, which are of course not possible in JavaScript.

Define and set default values for member variables that are not set by constructor parameters.

For example:

```
MyNamespace.Sample = function Sample(error, dataItems) {
    // call base constructor
    MyNamespace.Sample.initializeBase(this);
    // initialize member variables not set by constructor params
    this._handled = false;
    // initialize main properties, allowing for optional params
    this._error = error;
    this._dataItems = dataItems || {};
}
```

This approach helps to ensure that all class members are initialized correctly, and therefore reduces errors from confusing `undefined` and `null` (see section 5.5).

4.4.7 Event Design

The recommended mechanism for implementing Events is different depending upon the Ajax framework in use, due to the availability of the Sys.Observer class. Sys.Observer became available in the Ajax Control Toolkit. Both approaches are documented below; the Sys.Observer approach is preferred where available.

4.4.7.1 Manual Events

Events should be implemented in five stages;

1. Ensure the class has a member variable of type `Sys.EventHandlerList`.

```
this._events = new Sys.EventHandlerList();
```

This member variable is accessed via the method `get_events()` if your class derives from `Sys.Component`.

2. Create public methods to add and remove an event handler.

```
function add_amountChanged(handler) {
    this._events.addHandler("amountChanged", handler);
}
function remove_amountChanged(handler) {
    this._events.removeHandler("amountChanged", handler);
}
```

These handlers should expect two parameters – “sender” and “args”.

3. Optionally create a class derived from `Sys.EventArgs` to carry any data payload.

```
MyNamespace.AmountChangedEventArgs =
    function AmountChangedEventArgs(amount) {

    MyNamespace.AmountChangedEventArgs.initializeBase(this);
    this._amount = amount;
}

MyNamespace.AmountChangedEventArgs.prototype = {
    get_amount: function get_amount () {
        return this._amount;
    }
}

MyNamespace.AmountChangedEventArgs.registerClass(
    'MyNamespace.AmountChangedEventArgs', Sys.EventArgs);
```

4. Create a private method (prefixed by an underscore as per convention, see section 2.4.1) to raise each event. Use the raise terminology for events rather than fire or trigger. Allow the `EventArgs` derived class to be passed as a parameter, if required.

```
function _raiseAmountChanged(args) {
    if (!this._events) return;
    var handler = this._events.getHandler("amountChanged");
    if (handler) {
        handler(this, args);
    }
}
```

5. Call the raise method when the event should be fired.

```
if (someCondition){
    this._raiseAmountChanged(
        new MyNamespace.AmountChangedEventArgs(someParameter));
}
```

Use a derived class of `Sys.EventArgs` as the event argument if the event needs to send data to the event handler, or `Sys.EventArgs.Empty` if none is required.

Do not pass null as the sender or `EventArgs` parameter. By default use `this` and `EventArgs.Empty` respectively.

Be prepared for arbitrary code executing in the event-handling method.

4.4.7.2 Sys.Observer Events

To implement events using Sys.Observer there is significantly less work involved;

1. Create public methods to add and remove an event handler.

```
function add_amountChanged(handler) {
    Sys.Observer.addEventHandler(
        this, "amountChanged", handler);
}
function remove_amountChanged(handler) {
    Sys.Observer.removeEventHandler(
        this, "amountChanged", handler);
}
```

These handlers should expect two parameters – "sender" and "args".

2. Optionally create a class derived from `Sys.EventArgs` to carry any data payload, as described above.
3. When the class is disposed, ensure that all event handlers are removed;

```
dispose: function() {
    Sys.Observer.clearEventHandlers(this);
}
```

4. To raise an event, use the following syntax (passing in the optional EventArgs instance);

```
Sys.Observer.raiseEvent(this, "foo", args);
```

4.4.8 Event Handler Design

Define event handlers to expect two parameters;

```
function _handleAmountChanged(sender, args) {
    // code removed
}
```

When attaching a handler to an event from within a class instance, you may use the `createDelegate` method to ensure that the scope of the `this` keyword is as expected;

```
var handler = Function.createDelegate(this,
                  this._handleAmountChanged);
component.add_amountChanged(handler);
```

This ensures that “this” refers to the current class instance in the handleAmountChanged method.

It is recommended that event handlers are detached when no longer required.

4.4.9 Field Design

Prefix instance fields intended to be private with an underscore (see section 2.4.1). For simple data items that do not require logic to be executed during information retrieval or updating (e.g. notifying of data changes) use fields to keep download size small (see section 4.4.4). For more complex data items create properties that provide access to private data. This enables the implementation to be changed without affecting calling code.

4.5 Extensibility

Designing for extensibility leads to more flexible, reusable components, but can also lead to unnecessary overheads and complexity that are more obvious in JavaScript than other platforms. Therefore raw performance and simplicity must be balanced against extensibility and reuse.

When designing a library or reusable control, enhance the priority of extensibility.

The primary mechanisms for extensibility in the Ajax Control Toolkit are;

- Events; define events that other components can subscribe and react to.
- Base classes; define base classes to provide partial implementations, to simplify customization of behavior.
  Interfaces; use interfaces to indicate commonality between components, when base classes may already be chosen or there is no shared behavior.

4.6 Accessibility

An “accessible” web site means ensuring that it can be used by accessibility tools (such as Screen Readers), that it is device independent, and that it makes minimal assumptions about the users setup. This encompasses many topics, which are covered by the W3C Web Accessibility Initiative [10]. The ARIA specification is the latest authority on writing accessible internet applications [11].

There are a number of approaches that can assist with creating an accessible web site.

1. If your site must function without JavaScript, consider using Progressive Enhancement and Unobtrusive JavaScript. See section 4.2 for more information.
2. Ensure that event handlers do not assume the presence of a mouse. Consider using corresponding keyboard events in partnership with mouse events.
3. Provide non-script based alternatives for functionality. This can range from plain HTML downloads to telephone based services.
4. Test with Accessibility tools to ensure that functionality works well with them.

Further information is available from the W3C and WebAIM [12].

4.7 Localization

Messages, image locations, and other content displayed to a user via script may need to be localized as with any other web content.

Localized content should be placed in a separate script, named with a culture embedded in the filename. For example;

- MyScript.js
- MyScript.en-US.js
- MyScript.en-GB.js
- MyScript.es-CO.js

Further information on localization is available online [13].

4.8 Exceptions

Script errors should be defined using a function that uses the `Error.create` method to create an error instance, as follows;

```
MyNamespace.Errors.accountClosed =
    function accountClosed(sourceId) {

    var displayMessage = "MyNamespace.AccountClosedException: "
        + MyNamespace.Res.accountClosed;
    var err = Error.create(displayMessage, {
        name: "MyNamespace.AccountClosedException",
```

[10] Web Accessibility Initiative: http://www.w3.org/WAI/

[11] Accessible Rich Internet Applications Specification: http://www.w3.org/TR/wai-aria/

[12] WebAIM: http://www.webaim.org/techniques/javascript/

[13] Localization: http://www.asp.net/learn/Ajax/tutorial-04-cs.aspx

```
            source: sourceId
        });
        err.popStackFrame();
        return err;
    }
```

When an AccountClosedException is required, this method may then be called as follows:

```
    throw MyNamespace.Errors.accountClosed(this._id);
```

This implementation demonstrates;

- Ensuring Exceptions are created consistently using a function.
- Storing Exception messages using a localizable constant;

```
    MyNamespace.Res = {
        accountClosed: 'The account is already closed'
    }
```

- Passing the Exception's type name as a "name" parameter to Error.create;

```
    var err = Error.create(displayMessage, {
        name: "MyNamespace.AccountClosedException",
        source: sourceId
    });
```

- Using a call to `popStackFrame` to ensure the error creation function is not included in the reported stack.
- Optionally defining parameters specific to the Exception type. In this example, "sourceId" is specific to the AccountClosedException.

5 PROGRAMMING GUIDELINES

The topics in this section are intended to prevent developers from making common mistakes, or to highlight practices that can have a tangible effect upon performance, maintainability, or supportability.

5.1 Variable declaration

All variables should be declared using the "var" keyword to minimize the overheads in traversing the scope chain. Do not use global variables unless absolutely necessary as they are always the slowest to lookup. Variables should be initialized to a default value or explicitly set to null. For example:

```
    var counter = 0;
    var empty = null;
```

It is preferred to declare variables together at the top of a function. Group variables to minimize script size when possible, for example;

```
    var counter = 0, empty = null;
```

JavaScript does not have a notion of block scope; therefore variables defined within "`if`" or "`for`" blocks inside a function are accessible from anywhere within that function. This means it is not perceived a benefit to delay variable declaration until they are to be used. For example:

```
function doSomething() {
    var counter = 0, resultArray = [];

    // code removed
}
```

Avoid using the "with" statement. It reduces the clarity of code and modifies the scope chain, introducing performance penalties.

5.2 Function Shortcut Use

Avoid using function shortcuts such as `$get` within script that must coexist on a page with other scripts, or may do in the future. This is to avoid other scripts reassigning a global shortcut to an alternative function, which may cause your script to fail.

Instead use fully qualified function calls whenever a script may be reused in uncertain or shared circumstances, or for non-global functions use a closure as described in section 3.7.1.

5.3 Function Aliasing

When calling a function many times, such as in a loop, alias the function to reduce scope chain traversals. For example;

```
function doSomething() {
    var get = getDataByIndex;
    for (var counter = 0; counter < 10000; counter++) {
        var current = get(counter);
        // ...code removed
    }
}
```

Keep in mind that a function must be designed to be used in this way; for example, functions that rely on "`this`" may be unsuccessfully aliased.

5.4 Comparisons and Equivalence

Wherever possible use the strict equality operators to determine whether two values are equal. This prevents inefficient implicit type conversions and ensures that the responsibility for precision relies with the programmer. For example:

If “x” and “y” both have numeric values;
Wrong: `if (x == 5  && y != 4)`
Right: `if (x === 5 && y !== 4)`

5.5 Undefined and Null

Do not switch between using the `undefined` and `null` keywords. Ensure that all fields and variables are initialized to a value or `null`:

```
this._result = null;
```

Failing to do this would mean that `_result` would be `undefined`. When testing that a class instance variable is not set, or whether an optional parameter has been provided to a function, and the variable or parameter is known to be an Object, use the following syntax:

```
if (message) {
    // message was provided and not null
}
```

These two keywords are not equivalent, yet this approach reduces the occurrences of failed comparisons.

Finally, when explicitly testing whether or not a variable has been declared (such as checking whether a pageLoad function is defined, for example) use the following syntax:

```
if (typeof(pageLoad) === 'undefined') {
    // pageLoad has not been defined
}
```

This is a useful technique when it is necessary to detect the difference between undefined, null, false, zero, and empty string rather than treating all as equivalent.

5.6 Loops and Recursion

Loops and recursion can exaggerate the negative impact of poorly performing code. Therefore they should be a focus for good practices from the start, and optimized when deemed necessary. Some good practices are:

- Move expensive calls outside of the repeating code where possible. For example try/catch blocks and if-else branches that do not change during the loop can be extracted.

  Wrong:
  ```
  for (var counter = 0; counter < 10000; counter++) {
      try {
          performAction();
      } catch (e) {
          alert('Failure: ' + e);
          break;
      }
  }
  ```
  Right:
  ```
  try {
      for (var counter = 0; counter < 10000; counter++) {
          performAction();
      }
  } catch (e) {
      alert('Failure: ' + e);
  }
  ```

- Pre-calculate expensive loop exit conditions because they are evaluated for every iteration of the loop.

  Wrong:
  ```
  for (var counter = 0;
       counter < document.getElementsByTagName('div').length;
       counter++) {
      performAction();

  }
  ```
  Right:
  ```
  var target = document.getElementsByTagName('div').length;
  for (var counter = 0; counter < target; counter++) {
      performAction();

  }
  ```
  In this example, the call to `getElementsByTagName` would be recalculated from the DOM for every iteration. This is unlikely to be the intended behavior. Also ensure that `performAction` does not modify the loop collection.

- Use a single loop variable throughout functions that have multiple loops, to avoid declaring the same variable multiple times.

  Wrong:
  ```
  for (var counter = 0; counter < 10; counter++) {
      performAction();
  };
  ```

```
for (var counter = 0; counter < 10; counter++) {
    performOtherAction();
};
```

Right:

```
var counter;
for (counter = 0; counter < 10; counter++) {
    performAction();
};
for (counter = 0; counter < 10; counter++) {
    performOtherAction();
};
```

5.7 DOM Manipulation

5.7.1 Batch Up Changes

Manipulating the Document Object Model from JavaScript is expensive, and can result in requiring the browser to reflow the document. To reduce the number of DOM tree updates and screen repaints batch up changes and apply them at once.

Avoid concatenating HTML to the innerHTML property of elements (for example by using the += operator) as this potentially requires a large string concatenation before the DOM is modified.

5.7.2 Circular References

Circular references between DOM elements and JavaScript objects can lead to memory leaks in some older browsers. Although this issue is reduced in significance now, it should be considered for internet applications. Avoid storing references to JavaScript objects on expando properties on DOM objects as this reduces the likelihood of introducing memory leaks.

To understand more about memory leak patterns see "Understanding and Solving Internet Explorer Leak Patterns" [14].

5.8 Use of eval

The "eval" keyword invokes the JavaScript compiler and is therefore very slow. Avoid using eval where possible. Never use eval or the Function constructor on un-trusted data as this creates a potential script vulnerability. If eval is essential, use `window.eval` to execute in the global scope rather than just `eval`.

If using the ASP.NET Ajax Minifier ensure that you understand the "evals as safe" setting. Other minifiers may have equivalent settings.

[14] Understanding and Solving Internet Explorer Leak Patterns: http://msdn.microsoft.com/en-us/library/bb250448(VS.85).aspx