Script# Project Coding Guidelines

Nikhil Kothari, May 2007

This document provides a quick overview of coding guidelines in effect in the Script# project. Some of the guidelines concern stylistic aspects of the code, and few touch on actual code itself. They are generally based on the managed coding guidelines in the .NET Framework.

The intent of these guidelines is to ensure consistency in code across the project, to especially facilitate high degree of readability. The guidelines apply equally to public API/code as well as internal code.

# Code Organization

## Classes and Files

* Each file should contain at most one top-level class. Consider using nested classes if a class is only meaningful in the scope of another class (which is often the primary motivation for grouping classes into the same file).
* The class name and the file name should match.
* The file’s location on the file system should be indicative of the namespace that the class belongs to.
* Do not split code for a single class into multiple files using the partial classes. Partial classes are great for auto-generated code, and usage should be restricted to that. The one exception to this rule is for extremely big classes that are in fact a union of logically independent pieces of functionality. In this case, the file name should be <ClassName>.<LogicalGroup>.cs.

## File Layout and Member Order

* Namespace imports should occur top-level in the class (outside of the namespace scope), and should also be listed alphabetically, with the exception that System.\* namespaces should appear first. Also, do import the implicit System namespace, and use Visual Studio’s editor feature to remove unused namespaces.
* Members within a class should exist in the following order: Static fields and constants, member fields, constructors, properties, events, methods, explicitly implemented interface members, and finally nested classes.
* Member fields may be logically grouped, but other members (properties, events and methods) should be alphabetically ordered. Do not group by member visibility.
* Additionally, consider surrounding explicitly implemented interfaces with #region blocks (saying ‘Implementation of IFoo’ for example), so they can be collapsed in the code editor. The code editor automatically supports collapsing of classes, so this approach is not required for nested classes. Also, do not use #region blocks anywhere else in the code (eg. code blocks, or members etc).
* Do not initialize member fields at the point of their declaration. Initialize them in to the constructor. This makes sure the debugger does not jump around when debugging the instantiation of a class.
* Place a comment at the top of the file that indicates its name, and a path describing which project it belongs to.

Here is a sample code file

using System;

using System.ComponentModel;

using System.Diagnositics;

using System.Web;

using System.Web.UI;

namespace System.Web.UI.WebControls {

public class FooControl : WebControl, IAttributeAccessor {

private static string s\_imageUrl;

private const int \_imageSize = 16;

private Style \_imageStyle;

public FooControl() {

}

public Color ImageBackColor {

get {

object o = ViewState[“ImageBackColor”];

if (o != null) {

return (Color)o;

}

return Color.Empty;

}

set {

ViewState[“ImageBackColor”] = value;

}

}

public event EventHandler Click {

add {

}

remove {

}

}

public override void Render(HtmlTextWriter writer) {

}

#region Implementation of IattributeAccessor

string IAttributeAccessor.GetAttribute(string name) { ... }

void IAttributeAccessor.SetAttribute(string name, string value) { ... }

#endregion

private sealed class FooImage : Image {

// ...

}

}

}

# Style and Formatting

## Bracing Style

Use K&R style bracing where the open brace is on the end of the line. Also make sure there is a space between the last character and the open brace, and that the close brace matches the indentation of the line containing the open brace.

The following shows an example of the style applied to a set of constructs:

namespace System.Web.UI.WebControls {

public class WebControl : Control {

public WebControl() {

}

public Color ForeColor {

get {

object o = ViewState[“ForeColor”];

if (o != null) {

return (Color)o;

}

return Color.Empty;

}

set {

}

}

}

}

Braces should never be considered optional. Also split out single-line blocks into their own line rather than merging it with open/close braces on the same line.

for (int i = 0; i < 100; i++) {

DoSomething(i);

}

Else statements, Catch-blocks, Finally-blocks start on their own line.

try {

if (foo) {

// ...

}

else {

// ...

}

}

catch (Exception e) {

// ...

}

finally {

// ...

}

**Note**: If you use Visual Studio for your code editing, most of these settings are selected by default. You must however, choose to turn off the new-lines preceding open braces by going into Tools | Options; choosing the New Lines set of formatting options under the C# Text Editor group, and unchecking the 4 new lines options.

## Indentation

Indent each level by 4 spaces. Do not use Tabs. This ensures best readability across the board. Here is an example.

namespace System.Web.UI.WebControls {

public class WebControl : Control {

public Color ForeColor {

get {

object o = ViewState[“ForeColor”];

if (o != null) {

return (Color)o;

}

return Color.Empty;

}

set {

}

}

}

}

**Note**: Visual Studio 2005 actually defaults to this setting out-of-the-box… finally!

## Line Length

There is no fixed rule about line length. Just exercise good judgment and pick a reasonable line length. If the line seems too long, break it apart at logical spots such as parameters, the “=” in assignment statements etc. If a line exceeds 120 characters it is likely that it should be broken out.

## Spacing

Use a single space between arguments.

Wrong: streamReader.Read(buffer,0,100);

Right: streamReader.Read(buffer, 0, 100);

Do not use a space after and before parenthesis in a method call.

Wrong: streamReader.Read( buffer, 0, 100 );

Right: streamReader.Read(buffer, 0, 100);

Do not use spaces between the function name and the parenthesis.

Wrong: streamReader.Read (buffer, 0, 100);

Right: streamReader.Read(buffer, 0, 100);

Do not use spaces inside indexer-brackets.

Wrong: buffer[ index ];

Right: buffer[index]

Do use a single space after keywords in control-flow statements.

Wrong: while(x == y)

Right: while (x == y)

Do use a single space around operators.

Wrong: while (x==y)

Right: while (x == y)

## New Lines

* Do use a single new line as the separator between consecutive members.
* Do use two new lines as the separator between nested classes.
* Do use a single new line between the namespace declaration and the class declaration, and between the class declaration and its first member.
* Within a method, there is no set of explicit rules about when to use new lines and when not to. Use new lines to group together logically related sets of statements. In particular, do not introduce a new blank line between every consecutive pair of statements as a rule.

## Parenthesizing

Parenthesis around expressions involving binary operators in if, while statements are preferred. This improves readability, so one doesn’t have to think about operator precedence, and can focus on reading the code.

Wrong: while (x < y && x != 5)

Right: while ((x < y) && (x != 5))

Wrong: while (!boolValue && x < y)

Right: while (!boolValue && (x < y))

## Modifiers

Always explicitly indicate the access modifiers of classes and fields. Do not implicitly use the default “internal” or “private”.

**internal** class RenderUtilities {

**private** string \_name;

}

# Commenting

# General Comments

* Comments are meant for others reading/maintaining your code and for yourself when you are looking at your code after an extended period of time. This should guide what you comment, and what you put down in your comment.
* Do not re-state the obvious that can be inferred from reading the code. Strive to make sure your code is readable directly without requiring comments to explain the intent. At the same time, do add comments when they are likely to help.
* When you do decide to comment, place the comment on the line above the code. Do not place it to the right. Also make sure the comment stays with the code it is associated with, and is accurate as the code changes over time.
* Do not create block comments (using /\* and \*/) or some other unique mechanism to distinguish a set of comment lines. Each comment line should be preceded by “//”.

An example:

// add expando attributes

if (attrState != null) {

AttributeCollection atrColl = Attributes;

IEnumerator keys = atrColl.Keys.GetEnumerator();

while (keys.MoveNext()) {

string attrName = (string)(keys.Current);

writer.AddAttribute(attrName, atrColl[attrName]);

}

}

An even more useful comment

// Add expando attributes at the end, so that page developer’s attributes take

// precedence over comments we added.

if (attrState != null) {

...

}

## Markers

Do add comments to indicate temporary code, workarounds, or code that needs further review. Two standard prefixes have been chosen:

NOTE – Use “NOTE” for long-term comments.

TODO – Use “TODO” for short-term comments that indicate code that needs to be revisited, and possibly modified.

HACK – Use sparingly… for example if you’re adding some not-so-desirable code in response to an external behavior, i.e. unlikely to change anytime soon (i.e. not a TODO).

Do not put down your email address/name either.

// NOTE: This code should remain in sync with the corresponding logic in the Foo class.

// TODO: This is a temporary workaround to enable the Foo feature in beta 1.

# Naming

## Class, Interface and Member Names

Use Pascal casing (first letter of each word is capitalized) for type names, namespaces, and members of a type. In addition, for interfaces, use a leading “I”.

class HtmlTextWriter

interface IAttributeAccessor

void RenderContents(HtmlTextWriter writer)

## Field and Parameter Names

Use camel casing (first letter lowercase, first letter of each subsequent word capitalized) for parameter names, and member fields.

Use Pascal casing (first letter uppercase, first letter of each subsequent work capitalized) for static fields and constants.

class MyClass {

private int \_count;

public void DoSomething(int index, bool throwOnError) { ... }

}

## Leading Underscores

* Do not use a leading underscore for constants or statics.
* Do use a leading underscore for private members.

class MyClass {

public static readonly MyClass Default = new MyClass();

private static readonly object PropertyChangedEventKey = new object();

private const int MaximumValue;

private int \_count;

}

## Capitalization

Do not use capitalization for acronyms unless it is two letters at most. The exception to this is C# code that will be converted to script, where the norm is to use capital letters for the complete acronym (eg. encodeURIComponent).

**Note**: Use “ID” (and not “Id”). FxCop has gone back and forth on this, and hence this is called out here.

## Abbreviations

Do not use abbreviations in code, unless it is very established and commonplace (eg. UI)

## Hungarian Notation

Do not use hungarian. (The hungarian naming scheme uses the type and access prefixes before the variable name.)

Instead use good variable names that automatically indicate their type and intent and read better.

// Wrong

bool fVisible;

string strUserName;

int cUserName;

// Right

bool visible;

string userName;

int userNameLength;

# Implementation Considerations

## Choosing Access Modifiers

When you write a class, ask if it needs to be public. If not, make it internal. Do not create nested internal classes. Nested classes should always be private. Members on internal classes do not need to be marked internal – they should be public if they’re meant to be called from outside the class.

Generally you will want to write unsealed classes, so users may extend them. However, this is a decision that should be a part of the design process. When you write a class, ask if it makes sense to derive from, or if it makes sense to prevent derivation explicitly. Seal the class if it makes sense. You can always unseal the class in the future without incurring a breaking change.

When you write a method or property, ask if it makes sense to be overridable. If not, make it non-virtual. Methods and properties should be non-virtual by default. Like unsealed vs. sealed classes, deciding whether a member should be overiddable or not should be part of the design process. Also note that when you have overloaded methods, you want to choose one method with all the possible arguments as the candidate for making virtual, and not the rest.

Never create public or protected fields on any class. Keep in mind that a simple non-virtual property accessor does not have any associated runtime cost, and instead allows for further logic (such as validation) to be added in the future should it be needed. For writing C# code that will be converted to script, there are some scenarios where public fields are reasonable.

## String Handling

Always (unless you have a good reason to make an exception) treat String.Empty and null as the same value. Use String.IsNullOrEmpty(s) method for validation.

## Null Usage for Strings and Arrays

Never (unless you have a good reason for an exception) return null from a public API. Return String.Empty if the return type is a string; an empty array if the return type is an array.

This allows the caller to use the Length property without first checking for null.

It is fine to do so within internal code, where you do not want to incur the cost of creating an empty array.

## Argument Validation

Always perform argument validation. For public APIs throw an appropriate exception. For internal APIs perform a Debug.Assert.

For script code, perform argument validation only in debug builds.