StyleCop 4.1 Users Guide

**October 1, 2007**

# Contents

[Contents 1](#_Toc133595996)

[1.0 Overview 3](#_Toc133595997)

[2.0 Running StyleCop 3](#_Toc133595998)

[3.0 Exclusions 5](#_Toc133595999)

[4.0 Settings 6](#_Toc133596000)

[5.0 Rules 7](#_Toc133596001)

[5.1 C# Rules Analyzers 8](#_Toc133596002)

[5.1.1 Spacing 8](#_Toc133596003)

[5.1.2 Line Spacing 10](#_Toc133596004)

[5.1.3 Bracket Placement 11](#_Toc133596005)

[5.1.4 Xml Headers 13](#_Toc133596006)

[5.1.5 File Headers 14](#_Toc133596008)

[5.1.6 Element Order 14](#_Toc133596009)

[5.1.7 Declaration Keyword Order 17](#_Toc133596010)

[5.1.8 Class Members 17](#_Toc133596011)

[5.1.9 Variable Names 18](#_Toc133596012)

[5.1.10 Element name case 19](#_Toc133596013)

[5.1.11 Field Underscores 19](#_Toc133596014)

[5.1.12 Interface Names 20](#_Toc133596015)

[5.1.13 Access Modifiers 20](#_Toc133596016)

[5.1.14 Field Access Modifiers 20](#_Toc133596017)

[5.1.15 Comments 20](#_Toc133596018)

[5.1.16 Tabs 21](#_Toc133596019)

[5.1.17 Parenthesis 21](#_Toc133596020)

[5.1.18 Statements 21](#_Toc133596021)

[5.1.19 Method Parameter Placement 21](#_Toc133596022)

# 1.0 Overview

Historically, different development groups have used drastically different coding styles. Many teams have used inconsistent coding styles within a single product or even a single source file. StyleCop was originally written to provide a simple and efficient way to enforce a common coding style for C# code throughout Microsoft.

 Over time, StyleCop evolved to include new rules that go beyond style checks. StyleCop has become a good complement to FxCop (Code Analysis). FxCop analyzes compiled .Net binaries, while StyleCop analyzes the original source code. This allows StyleCop to investigate issues in code that is thrown away by the compiler. StyleCop could also be used to investigate issues in non-compiled languages such as JavaScript or Xml.

StyleCop provides an extensibility mechanism allowing parsers and analyzers for any language to be plugged into the system. Individual developers or teams can write and deploy their own rules, or integrate support for new languages. The StyleCop SDK provides support for writing StyleCop add-ins.

# 2.0 Running StyleCop

StyleCop can be run in a number of different ways. The tool can be installed on a developer’s machine and run within Visual Studio, providing menu items to perform analysis on a code file, project, or solution. An MSBuild task is also available, allowing teams to integrate StyleCop into any MSBuild-based build process such that it will be run automatically when a build is performed. The MSBuild task is discussed further in a separate document. A command-line executable is available, allowing a developer to run the tool against a file or a set of files. It is also possible to write a simple wrapper for StyleCop, to host it in virtually any build or development environment.

The StyleCop package for Visual Studio only runs under Visual Studio 2005 or later. It does not run under Visual Studio .Net 2003 or earlier versions of the IDE.



Figure 1. Running StyleCop within Visual Studio

To run StyleCop against all the files in the loaded solution, select *Tools/Run StyleCop* from the main menu. When run, StyleCop adds progress information to the Output pane, and writes violations to the StyleCop window:



Figure 2. StyleCop violations list

Double-clicking on a violation brings up the code at the point where the problem occurs.

Selecting the *Run StyleCop (Rescan All)* menu item causes StyleCop to perform a “clean analyze” against the selected files, ignoring cached results from previous runs.

To run StyleCop against the files in a single project, right-click on the project node and select *Run StyleCop*:



Figure 3. Running against a project.

You can also run the tool against a single file by right-clicking the file in Solution Explorer, or by bringing up the code document, and right-clicking directly on the document itself.

# 3.0 Exclusions

**Generated Code**

StyleCop can be prevented from running against code that was auto-generated by a tool. This should only apply to code that can be regenerated at a later time by the tool. In this case, it is a waste of time for the user to fix issues in the code, since these changes will just be overwritten later by the tool.

There are two ways to cause StyleCop to ignore blocks of generated code. For C# code, it is possible to use an *<auto-generated/>* header tag. This replaces the standard C# file header. The auto-generated header should look like:

//-----------------------------------------------------------------------

// <auto-generated>

// Provide a summary about the contents of the file and about the tool

// that generated the code.

// </auto-generated>

//-----------------------------------------------------------------------

This can be simplified down to:

// <auto-generated/>

A more descriptive header is recommended, however.

The second method for marking auto-generated code in C# is to use a generated code region. Place a *#region* directive around the area of generated code, and add directive text ending with the words “generated code”. For example:

#region Component Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

}

#endregion

**NativeMethods Classes**

*NativeMethods* classes typically contain element names which are derived directly from Win32 code. These names tend to violate modern naming guidelines. For this reason, StyleCop relaxes certain rules automatically for code within any *NativeMethods* class. This applies to and class ending with the name *NativeMethods*, including *SafeNativeMethods*, *UnsafeNativeMethods*, etc.

# 4.0 Settings

StyleCop provides a number of settings which modify the behavior of the tool. These settings are editable through the Visual Studio UI, and are saved on disk in a file called *Settings.StyleCop*. Settings are applied on a per-project basis. The *Settings.StyleCop* file is saved in the same folder as the project file.

The contents of the settings file are stored in an Xml format. The complete syntax is described at the end of this section.

It is possible to edit the settings for a project by right-clicking the project node in Visual Studio and choosing the *StyleCop Settings* menu item:



Figure 4. Editing settings

**Merged Settings**

It is also possible to create settings files which apply to multiple projects. The values in these ‘global’ settings files will be merged together with the values found in the project’s settings file. The way in which StyleCop discovers and merges settings files together is controlled through the *MergeSettingsFiles* property. There are three possible settings for this property:

1. *NoMerge*: Will not merge the settings with any other settings files.
2. *Parent*: Merges with a settings file found in a parent directory, if one exists. This is the default.
3. *Linked*: Merged with a specific settings file located at the path pointed to by the *LinkedSettingsFile* property.

If the value is set to *Parent*, or if the property is not present, StyleCop will attempt to locate a settings file in a parent folder of the current settings file, and will merge the settings from the two files together as follows:

1. Find the settings file closest to the root directory, and load the settings from that file. This will be known as the ‘parent’ settings file.
2. Move down the directory structure towards the project folder, and find the next settings file. This will be known as the ‘child’ settings file. If one exists, merge the settings from the child settings file into the settings from the parent settings file, as follows:
	1. If a property exists only in the parent settings file or only in the child settings file, use the value of the property as-is.
	2. If a property exists in both settings files and the values of the properties can be merged, merge the values. This chiefly applies to settings that contain lists of values.
	3. If a property exists in both settings files and the values of the properties cannot be merged, use the value from the child settings file.
3. The merged settings now become the new ‘parent’ settings file. Repeat step 2.

This feature can be used to easily control the settings for large portions of a source tree. For example, a team may choose a particular StyleCop setup for all production code. It can then place a *Settings.StyleCop* file at the root folder of all production code, containing those settings. The team may desire to use a different group of settings for all test code. It can then place a different settings file at the root of all test projects. In either case, individual projects can have their own local settings files, which override any settings in the parent settings files.

This setting can be edited through the Visual Studio UI by bringing up the StyleCop settings dialog for a project as described above, and then switching to the *Settings Files* tab. Using this dialog, it is also possible to edit the settings of a parent settings file or a linked settings file:



Figure 5. Settings files page

**Enabled Analyzers**

Using the settings files, it is possible to enable or disable a particular set of analyzers or language parsers. Some analyzers also expose additional properties which can be set in the settings files. The control these settings through the Visual Studio UI, change to the *Analyzers* tab:



Figure 6. Analyzers and detailed settings

Settings which apply to individual analyzers or parsers will be described in the rules section, below.

**Settings File Schema**

<StyleCopSettings Version="4.1">

 <GlobalSettings>

 <StringProperty Name="MergeSettingsFiles">*{NoMerge, Parent, Linked}*</StringProperty>

 <StringProperty Name="LinkedSettingsFile">*{path to file}*</StringProperty>

 </GlobalSettings>

 <Analyzers>

 <Analyzer AnalyzerId="*{analyzer ID}*">

 <AnalyzerSettings>

 <BooleanProperty Name="Enabled">*{True, False}*</BooleanProperty>

 *{other properties}*

 </AnalyzerSettings>

 </Analyzer>

 *{other Analyzer tags}*

 </Analyzers>

</StyleCopSettings>

The following is an example of a settings file which prevents merging with any other settings files, disables the LineSpacing analyzer, and sets some properties on the Documentation analyzer:

<StyleCopSettings Version="4.1">

 <GlobalSettings>

 <StringProperty Name="MergeSettingsFiles">NoMerge</StringProperty>

 </GlobalSettings>

 <Analyzers>

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="PublicAndProtectedOnly">True</BooleanProperty>

 <BooleanProperty Name="RequirePeriod">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

 <Analyzer AnalyzerId="StyleCop.CSharp.LineSpacing">

 <AnalyzerSettings>

 <BooleanProperty Name="Enabled">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

 </Analyzers>

</StyleCopSettings>

# 5.0 Parsers

A parser is a code module that reads code written in a particular language, and breaks that code down into an object model which can then be compiled or analyzed. StyleCop requires a special type of parser which provides details about all of the information within the source code, including whitespace, comments, etc. StyleCop analyzers read the information provided by the parser, and use it to check and enforce coding rules.

Currently, StyleCop ships with a parser for C#. Parsers for other languages are under development and may appear in the future.

## 5.1 C# Parser

The C# parser exposes one property which can be set by the user. This property determines whether Visual Studio designer files should be analyzed. These are special files which contain the text “.designer.” within the file name. Typically, certain sections of these files are auto-updated by Visual Studio during the development process. However, in many cases there are parts of the file which are written by a developer, or are safe for a developer to change.

By default, designer files are analyzed by StyleCop. If for some reason this needs to be prevented, this can by disabled by unchecking the following property:



Figure 7. Disabling analysis of designer files.

This property can also be set directly within a settings file:

 <Parsers>

 <Parser ParserId="StyleCop.CSharp.CsParser">

 <ParserSettings>

 <BooleanProperty Name="AnalyzeDesignerFiles">False</BooleanProperty>

 </ParserSettings>

 </Parser>

 </Parsers>

# 6.0 Rules Analyzers

StyleCop provides an extensibility model, which allows new rules to be written and plugged into the tool. See the StyleCop SDK for more information. This document describes the default rules which ship with the tool.

## 6.1 C# Rules Analyzers

### 6.1.1 Spacing

ID: StyleCop.CSharp.Spacing

* **Do** place a space after the following keywords when used to begin a statement:

*if*

*for*

*foreach*

*while*

*do*

*switch*

*catch*

*throw*

*lock*

*using*

*return*

*yield*

*new*

*stackalloc*

*fixed*

Example: if (true) ...

* **Do not** place a space after the following keywords when used to begin a statement:

*checked*

*unchecked*

*sizeof*

*typeof*

*default*

Example: Type type = typeof(int);

* **Do not** place a space after the name of a *method* when performing a method-call.

Example: int i = this.MyMethod(true);

* **Do** place a single space after a comma or semicolon, but no space before.

Example: for (int i = 0; i < 2; i++) ...

* **Do** place a single space before and after all arithmetic and relational symbols.

Example: if (i < 0 && i + 2 > 3) ...

* **Do** place a single space before, but no space after, all unary logical symbols.

Example: bool x = !y;

Example: int x = -y;

* **Do not** place a space after an opening parenthesis, square bracket, or generic bracket.
* **Do not** place a space before a closing parenthesis, square bracket, or generic bracket.

Example: int i = ((int)value);

Example: int i = value[2];

Example: int i = new Dictionary<string, List<int>>();

* **Do** place a single space after an opening curly bracket.
* **Do** place a single space before a closing curly bracket.

Example: int i = new int[] { 1, 2, 3 };

* **Do** place a single space before and after a colon base-clause colon.

Example: public class MyClass : BaseClass

* **Do** place a single space before and after a where-clause colon.

Example: public class MyClass<T> : BaseClass where T : int

* **Do** place a single space after, but no space before, a label or case-colon.

Example: case 2: return false;

* **Do** place a single space after, but not before, a dereference or address-of symbol when used in a type declaration.

Example: int\* x;

* **Do** place a single space before, but not after, a dereference or address-of symbol when used outside of a type declaration.

Example: object x = \*y;

* **Do not** place spaces between dereference and address-of symbols when more than one is specified in a row.

Example: int\*\*\* x;

* **Do not** place a space before a nullable-type symbol.

Example: int? x;

* **Do not** place spaces on either side of a member access symbol.

Example: this.data.Value.Count

* **Do** place a single space after a single-line comment symbol or an Xml documentation symbol.

Example: // This is a comment.

Example: /// <summary>

* **Do not** place a space after a preprocessor symbol.

Example: #region

* **Do not** place more than one space in a row outside of comments and strings.

### 6.1.2 Line Spacing

ID: StyleCop.CSharp.LineSpacing

* **Do not** place a blank line after an opening curly bracket or before a closing curly bracket.
* **Do not** place a blank line before an opening curly bracket.

Example: if (value)

{

 x += 1;

}

* **Do** place a blank line after a closing curly bracket when used to close a statement or element which spans multiple lines.

Example: if (value)

{

 x += 1;

}

y = 2;

* **Do not** place a blank line before an *else, catch,* or *finally* keyword, or before the *while* keyword in a *do/while* statement.

Example: ...

}

else

{

 x += 2;

}

* **Do not** place a blank line after a single-line comment or an Xml header.

Example: // Add the screen buffer.

width += buffer;

* **Do** place a blank line before an Xml header, unless the header is the first item in its scope.
* **Do** place a blank line before a single-line comment, unless it is immediately preceded by another single-line comment, or it is the first item in its scope.
* **Do not** place two or more blank lines back-to-back.

### 6.1.3 Curly Brackets

ID: StyleCop.CSharp.CurlyBrackets

* **Do not** omit curly brackets, even if the language allows it.

C# allows *if, while, for, foreach,* and *do* statements to be written with or without curly brackets. However, StyleCop requires brackets to be used in all cases. This increases code readability and maintainability. For example:

if (value)

{

 this.DoSomething();

}

* **Do not** place bracketed statements all on a single line.

Statements which contain bodies wrapped in curly-brackets should never be placed completely on a single line. For example, the following is not allowed:

if (value) { this.DoSomething(); }

As an exception, property, indexer, and event accessors are allowed to be completely on a single line if they are very small. For example:

public int Value

{

 get { return this.value; }

}

* **Do** place accessors completely on a single line if the other accessors in the same property, indexer, or event are also completely on a single line.

The following code is not allowed, since one accessor is written completely on a single line, and the other is not. In this case, both accessors should be written across multiple lines:

public int Value

{

 get { return this.value; }

 set

 {

 if (this.check == true)

 {

 return 1;

 }

 else

 {

 return 2;

 }

 }

}

* **Do** place opening and closing curly brackets on a line by themselves.

Opening and closing curly brackets must not share a line with other code unless the entire statement is written on a single line. The following bracket style must be used:

if (value)

{

 this.DoSomething();

}

The following are not allowed:

if (value) {

 this.DoSomething();

}

if (value)

{

 this.DoSomething(); }

This rule also applies to *do/while* statements, and *try/catch* statements, which must be written as:

do

{

 this.DoSomething();

}

while (value != null);

and:

try

{

 this.DoSomething();

}

catch (Exception)

{

 this.DoSomethingElse();

}

### 6.1.4 Documentation

ID: StyleCop.CSharp.Documentation

* **Do** place an Xml documentation header above every element in the file and write descriptive help text for all parts of the header.

Visual Studio provides built-in mechanisms to convert Xml headers into SDK documentation. These headers are also used at design-time to provide Intellisense information. For example, consider the following static method, which writes a value to the registry:

public class RegistryUtils

{

/// <summary>

/// Adds or overwrites a value under the HKCU key.

/// </summary>

/// <param name="name">The path to the value.</param>

/// <param name="valueobj">The object to write.</param>

/// <returns>Returns true if the value was written, false if not.</returns>

public static bool SetValue(string name, object valueobj)

{

 // write the value

}

}

When this method is called from elsewhere in the code, Visual Studio displays information in the form of a tooltip. The data in the tooltip is taken directly from the Xml header above the element:



***Settings:***

This rule may be configured by editing the settings for a project:



Figure 8. Documentation settings

*Public and protected only*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="PublicAndProtectedOnly">True</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

If this property is set, StyleCop will only require documentation on elements which are exposed outside of the assembly. By default this property is not set.

*Include fields*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="IncludeFields">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

If this property is set to False, StyleCop will not require documentation on fields.

*Require value tags*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="RequireValueTags">True</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

If this property is set to True, StyleCop will require a *value* tag for every property which is exposed outside of the assembly. These *value* tags are used within the SDK documentation that is produced by the header information. For example, consider the following SDK documentation for the Control.Font property:

**Control.Font Property**

Gets or sets the font of the text displayed by the control.

**public virtual** [Font](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemdrawingfontclasstopic.asp) **Font {get; set;}**

#### Property Value

The [Font](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemdrawingfontclasstopic.asp) object to apply to the text displayed by the control. The default is the value of the [DefaultFont](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemwindowsformscontrolclassdefaultfonttopic.asp) property.

The data displayed under the *Property Value* section is taken directly from the *value* tag in the Xml header for the property.

*Formatting and grammar*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="RequireProperFormatting">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

This property causes StyleCop to validate that all documentation text appears to be composed of valid English language sentences. This includes:

* The text is not empty.
* The text is at least 10 characters long.
* There are spaces between words.
* At least 40% of the characters are letters.

*End with a period*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="RequirePeriod">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

If this property is set to True, StyleCop will require every section of documentation text to end with a period.

*Start with a capital letter*

 <Analyzer AnalyzerId="StyleCop.CSharp.Documentation">

 <AnalyzerSettings>

 <BooleanProperty Name="RequireCapitalLetter">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

If this property is set to True, StyleCop will require every section of documentation text to begin with a capital letter.

### 6.1.5 File Headers

ID: StyleCop.CSharp.FileHeaders

* **Do** include a header at the top of every file C# file.

The file header must begin on the first line of the file. It must contain a *copyright* tag and optionally must contain a *summary tag.* The *copyright* tag must possess a *file* attribute containing the name of the file, and a *company* attribute containing the name of the company. The header must follow the given format:

//-----------------------------------------------------------------------

// <copyright file="FileName.cs" company="MyCompany">

// Copyright (c) MyCompany Corporation. All rights reserved.

// </copyright>

// <summary>Describes the contents of the file.</summary>

//-----------------------------------------------------------------------

The file headers rule may be configured to require a summary tag. This is not required by default.



Figure 9. File headers settings

*Require summary*

 <Analyzer AnalyzerId="StyleCop.CSharp.FileHeaders">

 <AnalyzerSettings>

 <BooleanProperty Name="RequireSummary">True</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

### 6.1.6 Element Order

ID: StyleCop.CSharp.ElementOrder

* **Do** place elements in a specific order, as defined below:

If the file contains a *namespace* element, this should be the first and only element at the root of the file. A file must not contain more than one *namespace* element.

The order of elements within a *namespace* element must be as follows. If the file does not contain a namespace element, this order applies at the root-level:

*Using directives*

*Enums*

*Interfaces*

*Structs*

*Classes*

Within a *class, interface,* or *struct*, elements must be written in the following order:

*Constants*

*Fields*

*Constructors*

*Destructors*

*Delegates*

*Events*

*Enums*

*Interfaces*

*Properties*

*Indexers*

*Methods*

*Structs*

*Sub-classes*

* **Do** order elements of the same type by their access, as follows:

*public*

*internal*

*protected internal*

*protected*

*private*

* **Do** place const and readonly elements before non-const, non-readonly elements of the same type.
* **Do** place static elements before non-static elements of the same type and access.
* **Do** place *using directives* within the *namespace* element.
* **Do not** place more than one *class* within a file. Multiple *partial classes* of the same type may be placed in the same file. Multiple sub-classes may be placed within a parent class.
* **Consider** implementing interfaces within a separate *partial class*. This allows the interface members to be placed next to one another. For example:

/// <summary>

/// My special class.

/// </summary>

public partial class MyClass

{

 // This class contains all of the custom code for MyClass.

}

/// <content>

/// Implements the IEnumerator interface.

/// </content>

public partial class MyClass : IEnumerator

{

 public object Current

 {

 get { return null; }

 }

 public bool MoveNext()

 {

 return false;

 }

 public void Reset()

 {

 }

}

By default, StyleCop applies the element order rules to generated code blocks as well as code written by hand. However, this setting can be disabled if the project requires it:



Figure 10. Element order settings

*Include generated code*

 <Analyzer AnalyzerId="StyleCop.CSharp.ElementOrder">

 <AnalyzerSettings>

 <BooleanProperty Name="GeneratedCodeElementOrder">False</BooleanProperty>

 </AnalyzerSettings>

 </Analyzer>

### 6.1.7 Keyword Order

ID: StyleCop.CSharp.DeclarationKeywordOrder

* **Do** place declaration keywords in a specific order, as defined below:

In an element declaration, the access modifier should always appear first. If the element is static, const, etc., these keywords should appear next, followed by the element type or name. For example, the following element declarations are correct:

private int customerAge;

private static int customerAge;

private static readonly int customerAge;

But the following is incorrect because the access modifier should come at the beginning of the declaration.

static private int customerAge;

* **Do** place the keyword *protected* before the keyword *internal* for elements with protected internal access.

### 6.1.8 Class Members

ID: StyleCop.CSharp.ClassMembers

* **Do** prefix calls to class members with *‘this.*’.

All calls made to other members of the same class, including *fields, properties, methods, indexers, events,* etc., must be prefixed by the token *this* to indicate that the referenced item lives within the same class. This also applies to members contained within base classes of the current class.

* **Do not** prefix calls to base class members with ‘*base.*’, unless a base class member is being called from an override of the same member in the local class.

Prefixing base-class members with ‘*base.*’ can potentially lead to maintainability problems. If the base-class member is later made virtual, and an override is created in the local class, the local override will not be called since the local call is explicitly prefixed by ‘*base.*’. For example:

public class BaseClass

{

 protected void BaseMethod()

 {

 }

}

public class DerivedClass : BaseClass

{

 public void MyMethod()

 {

 base.BaseMethod();

 }

}

If the code is changed to:

public class BaseClass

{

 protected virtual void BaseMethod()

 {

 }

}

public class DerivedClass : BaseClass

{

 public void MyMethod()

 {

 base.BaseMethod();

 }

 protected override void BaseMethod()

 {

 }

}

The local override of BaseMethod will not be called. Instead, the call to BaseMethod() should be prefixed with ‘*this.*’ to indicate that it part of the local instance.

The ‘*base.*’ prefix may only be used within an overriden method or property, when calling the base implementation of that same method or property. For example:

public class BaseClass

{

 protected virtual void BaseMethod()

 {

 }

}

public class DerivedClass : BaseClass

{

 protected override void BaseMethod()

 {

 base.BaseMethod();

 }

}

### 6.1.9 Naming

ID: StyleCop.CSharp.Naming

* **Do** use Pascal casing when naming constant *fields* and *variables*.
* **Do** use Pascal casing when naming readonly *fields* and *variables* which are not declared *private*. Private readonly *fields* and *variables* may be cased with Pascal or camel casing.
* **Do** use camel casing when naming non-constant, non-readonly *fields* and *variables*.
* **Do** use Pascal casing when naming *public* or *internal* fields.
* **Do** use pascal casing (beginning with an upper-case letter) when naming the following element types: *namespaces, classes, structs, enums, delegates, events, properties, methods.*
* **Do not** prefix *fields* in a *class* or *struct* with ‘*m\_*’.
* **Do not** use underscores within *field* names.
* **Do** begin the names of all *interfaces* with the capital letter ‘I’.
* **Do not** use Hungarian notation when naming *fields* and variables.

StyleCop considers any any variable name that starts with one or two lower-case characters followed by an upper-case character to be Hungarian notation. In actuality, it is possible to specify one or two character sequences which are not Hungarian notation, and which should be allowed. To mark prefixes as acceptible and force StyleCop to accept them, bring up the settings dialog for a project and change to the *Hungarian* tab:



Figure 10. Allowed prefixes

StyleCop will ignore any prefix that appears in this list. This can also be set directly within the settings file:

 <Analyzer AnalyzerId="StyleCop.CSharp.Naming">

 <AnalyzerSettings>

 <CollectionProperty Name="Hungarian">

 <Value>is</Value>

 <Value>on</Value>

 </CollectionProperty>

 </AnalyzerSettings>

 </Analyzer>

### 6.1.10 Access Modifiers

ID: StyleCop.CSharp.AccessModifiers

* **Do** explicitly declare access modifiers on all elements.

Every element declaration should begin with an access modifier keyword; either *public, private, protected*, or *internal*. In C# it is legal to declare an element without an access modifier, in which case the protection of the element defaults to *private*. However, to maintain consistency and transparency in the code base, StyleCop requires access modifiers to be explicitly declared on all elements.

* **Do** set the access of all class fields to *private.* Use properties to expose fields outside of the *class*. Within a struct, fields may have any access level. Constant fields may have any access type.

### 6.1.11 Comments

ID: StyleCop.CSharp.Comments

* **Do not** write “empty” single-line comments unless they fall within the middle of a block of comments.

Invalid: //

// The empty line above is invalid, but the empty line below is ok

// since it falls in the middle of this block of comments.

Valid: //

// The empty line below is also invalid.

Invalid: //

int x = 0;

* **Do not** write empty comments using the multi-line comment style. For example:

/\*

\*/

* **Do not** use the triple-slash comment style for single-line comments. This style is reserved for Xml documentation headers.
* **Do** use four slashes in a row when commenting out lines of code. This distinguishes out-commented code from descriptive comments. Four slashes must be used since three slashes are reserved for Xml documentation headers.

 // This is a normal comment.

 CsDocument csdocument = document as CsDocument;

 Debug.Assert(csdocument != null);

 if (csdocument != null)

 {

 ////if (csdocument.RootElement != null)

 ////{

 //// this.ProcessComments(csdocument.RootElement);

 ////}

 }

### 6.1.12 Tabs

ID: StyleCop.CSharp.Tabs

* **Do not** use tabs for indentation, or anywhere else within a code document. Each level of indentation should consist of four spaces.

### 6.1.13 Parenthesis

ID: StyleCop.CSharp.Parenthesis

* **Do not** use parenthesis in places where they are unnecessary and do not increase readability.

In the following example, the parenthesis are unnecessary and should be removed:

return ((this.data).Value).Count;

###

### 6.1.14 Statements

ID: StyleCop.CSharp.Statements

* **Do not** place more than one statement on a single line.
* **Do not** add an extra semicolon to the file, resulting in an empty statement.

### 6.1.15 Method Parameter Placement

ID: StyleCop.CSharp.MethodParameters

* **Do** place the opening parenthesis of a method call on the same line as the method name.
* **Do** place the opening square bracket of an array access on the same line as the item name.
* **Do** place the closing parenthesis of a method call on the same line as the last parameter. If there are no parameters, place the closing parenthesis on the same line as the opening parenthesis.
* **Do** place the closing square bracket of an array access on the same line as the last array index parameter.
* **Do** place a comma separating two method parameters on the same line as the previous method parameter.

* **Do** place all method parameters on the same line, or else place each parameter on its own line.
* **Do** place each parameter on the same line as the previous parameter (if all parameters are on the same line), or else on the line immediately following the previous parameter.
* **Do** place the first parameter on the line after the method name when the parameters are on separate lines.

StyleCop allows method parameters to be laid out in one of three ways:

MyMethod(item1, item2, item3, item4);

MyMethod(

 item1, item2, item3, item4);

MyMethod(

item1,

item2,

item3,

item4);

The following styles violate the above rule and are not allowed:

MyMethod(item1, item2,

 item3, item4);

MyMethod(

 item1, item2,

 item3, item4);

MyMethod(item1,

 item2,

 item3,

 item4);

MyMethod(

 item1, item2, item3, item4);

MyMethod(

 item1,

 item2,

 item3,

 item4);

* **Do not** allow any parameter other than the first parameter to span across multiple lines.

The following method calls are legal because only the first parameter spans across multiple lines:

MyMethod(string.Format(

 “Fill {0} the {1}”,

 item1,

 item2));

MyMethod(new int[]

{

 1, 2, 3

});

However, the following call is illegal because the second and third parameter spans across multiple lines. Because of this, this method call is difficult to read.

MyMethod(

 item1,

 "This is a string " +

 "which continues onto a second line",

 new int[]

 {

 1, 2, 3

 },

 5);

The second and third parameters should be stored in local variables and then passed into the method call. This makes the call much more readable.

string text = "This is a string " +

 "which continues onto a second line";

int[] values = new int[]

{

 1, 2, 3

};

MyMethod(item1, text, values, 5);