Coding Conventions - C#

Short and simple conventions for C# code.

```
/// <summary>
/// Evaluates the respect of conventions.
/// All sample code in this document respect these conventions.
/// </summary>
public class Conventions : IStudentEvaluator
{
    const bool _isOptional = false;

    /// <summary>
    /// Gets whether respecting condition is an option.
    /// Always false.
    /// </summary>
    public bool IsOptional
    {
        get { return _isOptional; }
    }

    /// <summary>
    /// Tests how the submitted code respects the conventions and
    /// impacts the student grade accordingly.
    /// </summary>
    /// <param name="code">Source code in C#. Must not be null.</param>
    /// <param name="g">Current evaluation of the source code (between 0 and 1.0).</param>
    /// <returns>Impacted grade (between 0 and 1.0).</returns>
    public double ImpactGrade( string code, double g )
    {
        if( code == null ) throw new ArgumentNullException( "code" );
        if( g < 0.0 || g > 1.0 ) throw new ArgumentException( "Must be in [0,1.0].", "g" );
        return RespectsAllConventions( code ) ? Math.Max( g * 1.1, 1.0 ) : 0.0;
    }
}
```

Contents

1 One file, one class

Each file should contain one top-level class. Consider using private nested classes if a class is only meaningful in the scope of another class. For large class, use partial definition and span the code across multiple files with a logical/functional suffix ("Big.cs", "Big.Runner.cs", "Big.Export.cs").

The class name and the file name should match.

Namespace imports should occur top-level in the class (outside of the namespace scope). Remember to use "Organize Usings > Remove and Sort" from time to time.

2 Class members organization

Members within a class should appear in the following order:

1. Fields (either static or instance).
2. Type initializer (static constructor) - If there is one.
3. Constructors.
4. Finalizer - If there is one.
5. Factory static methods if any.

All other members should be organized by logical/functional aspects regardless of their kind. Regions can be used to group such aspects. Explicitly implemented interfaces, nested types and any other constructs should appear where it makes the most sense.

Please, do not group by member visibility.

3 Visibility

In C# members are private and Types are internal by default. And this is good: like in the domain of security, it starts "closed" and becomes "opened" only if and when needed.

Avoid specifying such defaults (they are more, actually useless, words to read).

4 Constructors & Initialization

Initialize members within the constructor, and not at the point of their declaration.
Use "`: this( … )`" relay between constructors to enforce DRY principle as much as possible.

Use Type initializer as soon as static fields' initialization requires anything other than a simple assignment.

5 Casing

Use **PascalCasing** (first letter of each word is capitalized) for Type names, Namespaces, Properties, Methods, Events, and public Fields (either static or instance fields).

Use **camelCasing** (first letter lowercase, first letter of each subsequent word capitalized) for Parameters, Local variables.

Use underscore prefixed **_camelCasing** for private Fields (either static or instance fields).

Since protected fields are *de facto* publicly visible from other assemblies, use **PascalCasing**.

Internal fields are not visible, they can use **_camelCasing**, but it is always better to expose an internal Property bound to a private _field in this case (avoid exposing an *internal field*, even to your own code!).

```
using System;

namespace Intech.Sample
{
    public class StupidClass
    {
        readonly static int _staticField;
        readonly int _field;
        int _mutableField;

        static SampleClass()
        {
            _staticField = DateTime.UtcNow.Millisecond;
        }

        public SampleClass()
        {
            _field = _staticField * DateTime.UtcNow.Millisecond;;
        }

        public SampleClass( int value )
            : this()
        {
            _mutableField = value;
        }

        public int Property
        {
            get { return _mutableField; }
            set { _mutableField = value; }
        }

        public int ReadOnlyProperty
        {
            get { return _field * _mutableField; }
        }

        public void Method()
        {
            _mutableField = 0;
        }
    }
}
```

6 Use of aliased Type name

Always use C# keywords for primitive types (`string`, `int`, `double`, etc.) even you are referring to a static method or property (e.g. `string.Empty`, `int.MinValue`, `double.TryParse( … )`, etc.).

7 The this keyword

Do not use `this` to refer to a member: casing and _ prefix do the job.
For extension methods, use `@this` to name the “this” object:

```
/// <summary>
/// Strongly typed version of <see cref="IServiceProvider.GetService"/> that returns null if service
/// is not found. (Same behavior as <see cref="IServiceProvider.GetService"/>.)
/// </summary>
/// <param name="this">This service provider.</param>
/// <returns>A service object of the required type or null if not found.</returns>
public static T GetService<T>( this IServiceProvider @this )
{
    return (T)@this.GetService( typeof( T ) );
}
```

Do NOT check for null on `@this`. Any access to the null reference will throw it anyway and this unifies the programming API.

8 Indent style

Use the Allman style (http://en.wikipedia.org/wiki/Indent_style).

Conditional compilation symbols (`#if !net40 … #endif`) must be aligned with the code (not at the beginning of the line).

```
internal void FlushBuffer( Func<string,IChannel> newChannels )
{
    #if !net40
    Debug.Assert( Monitor.IsEntered( _flushLock ) );
    #endif
    Debug.Assert( _useLock.CurrentCount >= 1 );

    _useLock.Signal();
    _useLock.Wait();
    if( newChannels == null )
    {
        GrandOutputEventInfo e;
        while( _buffer.TryDequeue( out e ) );
    }
    else
    {
        GrandOutputEventInfo e;
        while( _buffer.TryDequeue( out e ) )
        {
            IChannel c = newChannels( e.Topic );
            c.Handle( e, false );
        }
    }
    Debug.Assert( _useLock.CurrentCount == 0 );
}
```

Do not use Tabs characters, use blank spaces instead with an indent size of 4.

Visual Studio handles this automatically (except the conditional compilation symbols): see Use Visual Studio settings below.

9 Parens & white spaces

Avoid useless parentheses around returned value:

```
    get { return 4 * _flushLockCount - _otherCount; }
```

Always use parentheses around bitwise binary operators and for boolean expressions for which operators' priority may be ambiguous:

```
Debug.Assert( fake == null || ((level & LogLevel.IsFiltered) != 0 && MaskedLevel != LogLevel.None) );
```

Favor code on language artefacts by adding spaces around expressions:

```
if( startAt < maxLength - 2 && s[startAt] == '(' )
{
    int iStartNum = startAt + 1;
    int iCloseB = s.IndexOf( ')', iStartNum );
    if( iCloseB > 0 )
    {
        if( Byte.TryParse( s.Substring( iStartNum, iCloseB - iStartNum ), out uniquifier ) )
        {
            startAt = iCloseB + 1;
        }
    }
}
```

Just like indentations, Visual Studio handles this automatically: see Use Visual Studio settings below.

10 Comments

Use `///` comments, especially on public or protected Types and Members. Try to specify what could be important and could help the developer who will use the code (recall that the `<summary/>` is the tooltip and that the comment about the parameter is also displayed).

Use conventional comments for properties: "Gets the…" for read only properties, "Gets or sets the…" for writeable ones.

A comment refers to a thing: use verbs in the third person (Gets, Sets, Checks, Matches, Resolves, …).

```
/// <summary>
/// Gets the unique identifier for this monitor.
/// It is a <see cref="Guid.NewGuid"/> by default but specialized implementations can set it
/// via <see cref="SetUniqueId"/> if a unique identifier exists in the context that can
/// more easily identify this activity.
/// </summary>
protected Guid UniqueId
{
    get { return _uniqueId; }
}

/// <summary>
/// Gets or sets the tags of this monitor: any subsequent logs will be tagged by these tags.
/// The <see cref="CKTrait"/> must be registered in <see cref="ActivityMonitor.Tags"/>.
/// Modifications to this property are scoped to the current Group since when a Group
/// is closed, this property (like <see cref="MinimalFilter"/>) is automatically
/// restored to its original value (captured when the Group was opened).
/// </summary>
public CKTrait AutoTags { get { … } set { … } }
```

```
/// <summary>
/// Returns files in a directory according to multiple file masks (separated by ';').
/// </summary>
/// <param name="path">Path of the directory to read.</param>
/// <param name="multiFileMask">File masks, for example: *.gif;*.jpg;*.png.</param>
/// <returns>List of files' full name (without duplicates).</returns>
static public string[] GetFiles( string path, string multiFileMask )
{
    …
}
```

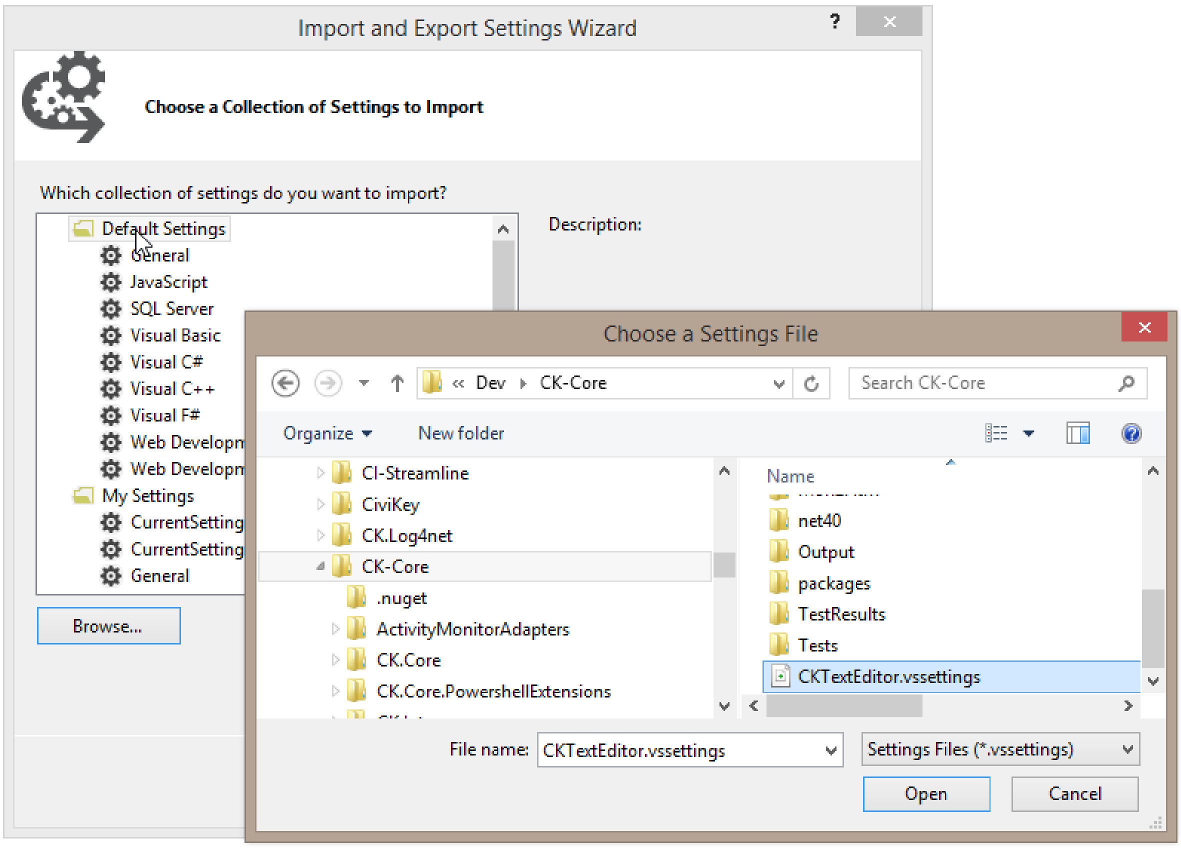
For public (and professional code), use xml elements to link and format the documentation: see Recommended Tags for Documentation Comments (C# Programming Guide).

Do not fully comment your code too early. Start with simple comments that goes straight to the fact and that will help you to use your own code. The more settled the code base is, the more you can develop and refine the comments.

11 Use Visual Studio settings

Integrated Development Environment like Visual Studio handles quite well auto formatting of source code.

Use this file https://github.com/Invenietis/ck-core/blob/master/CKTextEditor.vssettings to configure Visual Studio by using "Tools > Import and Export Settings". Get the file above and import it:


Import and Export Settings Wizard
Choose a Collection of Settings to Import
Which collection of settings do you want to import?
Default Settings
General
JavaScript
SQL Server
Visual Basic
Visual C#
Visual C++
Visual F#
Web Developm
Web Developm
My Settings
CurrentSetting
CurrentSetting
General
Browse...
Description:
Choose a Settings File
Dev
CK-Core
Search CK-Core
Organize
New folder
CI-Streamline
CiviKey
CK.Log4net
CK-Core
.nuget
ActivityMonitorAdapters
CK.Core
CK.Core.PowershellExtensions
Name
net40
Output
packages
TestResults
Tests
CKTextEditor.vssettings
File name:
CKTextEditor.vssettings
Settings Files (*.vssettings)
Open
Cancel