miloszkukla Update coding-style.md (#36695)

7b320a0 on Apr 9

12 contributors

134 lines (107 sloc) 6.37 KB

C# Coding Style

For C++ files (*.cpp and *.h), we use clang-format (version 3.6+) to ensure code styling. After changing any Cpp or H file and before merging, src/Native/format-code.sh must be run. This script will ensure that all native code files adhere to the coding style guidelines.

For non code files (xml, etc), our current best guidance is consistency. When editing files, keep new code and changes consistent with the style in the files. For new files, it should conform to the style for that component. If there is a completely new component, anything that is reasonably broadly accepted is fine.

The general rule we follow is "use Visual Studio defaults".

1. We use Allman style braces, where each brace begins on a new line. A single line statement block can go without braces but the block must be properly indented on its own line and must not be nested in other statement blocks that use braces (See issue 381 for examples). One exception is that a `using` statement is permitted to be nested within another `using` statement by starting on the following line at the same indentation level, even if the nested `using` contains a controlled block.
2. We use four spaces of indentation (no tabs).
3. We use `_camelCase` for internal and private fields and use `readonly` where possible. Prefix internal and private instance fields with `_`, static fields with `s_` and thread static fields with `t_`. When used on static fields, `readonly` should come after `static` (e.g. `static readonly` not `readonly static`). Public fields should be used sparingly and should use PascalCasing with no prefix when used.
4. We avoid `this.` unless absolutely necessary.
5. We always specify the visibility, even if it's the default (e.g. `private string _foo` not `string _foo`). Visibility should be the first modifier (e.g. `public abstract` not

`abstract public`).

6. Namespace imports should be specified at the top of the file, *outside* of `namespace` declarations, and should be sorted alphabetically, with the exception of `System.*` namespaces, which are to be placed on top of all others.
7. Avoid more than one empty line at any time. For example, do not have two blank lines between members of a type.
8. Avoid spurious free spaces. For example avoid `if (someVar == 0)...` , where the dots mark the spurious free spaces. Consider enabling "View White Space (Ctrl+E, S)" if using Visual Studio to aid detection.
9. If a file happens to differ in style from these guidelines (e.g. private members are named `m_member` rather than `_member`), the existing style in that file takes precedence.
10. We only use `var` when it's obvious what the variable type is (e.g. `var stream = new FileStream(...)` not `var stream = OpenStandardInput()`).
11. We use language keywords instead of BCL types (e.g. `int, string, float` instead of `Int32, String, Single` , etc) for both type references as well as method calls (e.g. `int.Parse` instead of `Int32.Parse`). See issue 391 for examples.
12. We use PascalCasing to name all our constant local variables and fields. The only exception is for interop code where the constant value should exactly match the name and value of the code you are calling via interop.
13. We use `nameof(...)` instead of `"..."` whenever possible and relevant.
14. Fields should be specified at the top within type declarations.
15. When including non-ASCII characters in the source code use Unicode escape sequences (\uXXXX) instead of literal characters. Literal non-ASCII characters occasionally get garbled by a tool or editor.
16. When using labels (for goto), indent the label one less than the current indentation.

An EditorConfig file (`.editorconfig`) has been provided at the root of the corefx repository, enabling C# auto-formatting conforming to the above guidelines.

We also use the .NET Codeformatter Tool to ensure the code base maintains a consistent style over time, the tool automatically fixes the code base to conform to the guidelines outlined above.

Example File:

`ObservableLinkedList`1.cs`:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
```

```
using System.ComponentModel;
using System.Diagnostics;
using Microsoft.Win32;

namespace System.Collections.Generic
{
    public partial class ObservableLinkedList<T> : INotifyCollectionChanged, INot
    {
        private ObservableLinkedListNode<T> _head;
        private int _count;

        public ObservableLinkedList(IEnumerable<T> items)
        {
            if (items == null)
                throw new ArgumentNullException(nameof(items));

            foreach (T item in items)
            {
                AddLast(item);
            }
        }

        public event NotifyCollectionChangedEventHandler CollectionChanged;

        public int Count
        {
            get { return _count; }
        }

        public ObservableLinkedListNode AddLast(T value)
        {
            var newNode = new LinkedListNode<T>(this, value);

            InsertNodeBefore(_head, node);
        }

        protected virtual void OnCollectionChanged(NotifyCollectionChangedEventAr
        {
            NotifyCollectionChangedEventHandler handler = CollectionChanged;
            if (handler != null)
            {
                handler(this, e);
            }
        }

        private void InsertNodeBefore(LinkedListNode<T> node, LinkedListNode<T> n
        {
           ...
        }

        ...
    }
}
```

ObservableLinkedList`1.ObservableLinkedListNode.cs:

```
using System;

namespace System.Collections.Generics
{
    partial class ObservableLinkedList<T>
    {
        public class ObservableLinkedListNode
        {
            private readonly ObservableLinkedList<T> _parent;
            private readonly T _value;

            internal ObservableLinkedListNode(ObservableLinkedList<T> parent, T v
            {
                Debug.Assert(parent != null);

                _parent = parent;
                _value = value;
            }

            public T Value
            {
               get { return _value; }
            }
        }

        ...
    }
}
```