Full Stack

Coding Standards & Conventions

Developers' Guide

Version 1
4-24-2017

Introduction 3
What are Coding Standards? 3
Scope 3
1. Naming Guidelines 4
1.1 Capitalisation Conventions 4
1.2 Capitalising Compound Words and Common Terms 5
2. General Naming Conventions 6
2.1 Word Choice 6
2.2 Using Abbreviations and Acronyms 6
3. Names of Namespaces 7
4. Names of Classes, Structs and Interfaces 8
4.1 Names of Generic Type Parameters 9
5. Names of Type Members 10
5.1 Names of Methods 10
5.2 Names of Properties 10
5.3 Names of Events 11
5.4 Names of Fields 11
6. Names of Parameters 12
7. Names of Unit Test Methods 12
8. Async/Await Patterns 13
8.1 Naming Convention 13
8.1 Avoid async void 13
8.2 Async all the way 13
8.3 Configure Context 14
9. Exception Handling 15
10. Other 16
10.1 Newlines/Whitespace 16
10.2 Ternary Statements 16
10.3 Superfluous Parameters 16
10.4 Superfluous Variables 16
10.5 Class per File 16
10.6 Floating-Point Comparisons 17
10.7 String Comparison 18
10.8 Pseudo-code & Code Comments 18

10.9 String Literals 19
10.10 Code Regions 19
10.11 Terse Getters & Setters 19
10.12 Expression Bodied Members 20

Introduction

At Full Stack we expect our developers to be the embodiment of our motto: be **smart** and get things **done**. This document directly addresses our motto in that it serves to suggest, steer and, at times, dictate better coding habits and practices so that Full Stack can be a cost effective, productive and efficient software development environment for all.

This is a living/breathing thing; we live in a world where new technologies frequently emerge so this will have to be reviewed from time to time to keep with the times.

What are Coding Standards?

Coding standards and conventions exist to help you write code that is robust, resistant to the most common and damaging errors and is based on decades of collective experience. This document strives to create uniformity across all codebases thereby eliminating confusion and establishing a common interpretation of the Full Stack coding standard and styles for all who work here, now and in the future.

Scope

Parts of this document is unashamedly adapted, and in some cases copied directly, from Microsoft's Framework Design Guideline and therefor, for the time being, only applies to the C# language and the .NET Framework Common Type System (CTS).

https://en.wikipedia.org/wiki/Common_Type_System

1. Naming Guidelines

1.1 Capitalisation Conventions

Capitalisation, when applied consistently, make identifiers for types, members, variables and parameters easy to distinguish and read.

To differentiate words in an identifier, capitalise each word in the identifier. There are only two appropriate ways to capitalise identifiers, depending on the use of the identifier

- PascalCasing
- camelCasing

PascalCasing is used for all identifiers except parameter names, private field names and local variable names.

The PascalCasing convention capitalises the first character of each word (including acronyms over two letters in length), as shown in the following examples:

- `FullName`
- `HtmlTag`
- `IosMobileApp`

A special case is made for two letter acronyms in which both letters are capitalised.

- `IOStream`
- `FHAccount`

The camelCasing convention, used for parameter and private field names, inline variables and local variables, capitalises the first character of each word except the first word, as shown in the following examples. The examples also show, two-letter acronyms that begin a camel-cased identifier are both lowercase.

- `fullName`
- `htmlTag`
- `iosMobileApp`
- `ioStream`
- `fhAccount`

Please note, private fields should be prefixed with underscores followed by camelCasing.

- `_privateField`
- `_idNumber`
- `_fhAccount`

1.2 Capitalising Compound Words and Common Terms

Compound terms are treated as single words for purposes of capitalisation. Do not capitalise each word in a so-called closed-form compound word. Use a current dictionary to determine if a compound word is written in closed form.

Here is a list of examples:

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	EndPoint
FileName	fileName	Filename
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Metadata	metadata	MetaData, metaData
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder
SignIn	signIn	SignOn
SignOut	signOut	SignOff
UserName	userName	Username
WhiteSpace	whiteSpace	Whitespace
Writable	writeable	Writeable

2. General Naming Conventions

This section describes general naming conventions that relate to word choice and guidelines on using abbreviations and acronyms.

2.1 Word Choice

Choose easily readable identifier names. For example, a property named `HorizontalAlignment` is more readable than `AlignmentHorizontal`. Favor readability over brevity. The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).

Do not use hyphens or any other non-alphanumeric characters. Do not use ***Hungarian notation***.

2.2 Using Abbreviations and Acronyms

Do not use abbreviations or contractions as part of identifier names. For example, use `GetCoordinates` rather than `GetCoords`. Do not use any acronyms that are not widely accepted, and if they are, only when necessary.

3. Names of Namespaces

Don't underestimate the importance of clear and well named namespaces. A well-conceived namespace name provides clarity and immediately indicates what the content of the namespace is likely to be.

The following template specifies the general rule for naming namespaces:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

Here are some examples to consider:

- `CompanyX.Mobile.Core`
- `CompanyX.Mobile.Ios`
- `CompanyX.Web.Controls`

Prefix namespace names with a company/client name to prevent namespaces from different companies/clients from having the same name. Use a stable, version-independent product name at the second level of a namespace name. Do not use organisational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organise the hierarchy of namespaces around groups of related technologies. Use PascalCasing, and separate namespace components with periods (e.g., `CompanyX.Mobile.Security`).

Consider using plural namespace names where appropriate, e.g. use `CompanyX.Collections` instead of `CompanyX.Collection`. However, acronyms are exceptions to this rule. For example, use `System.IO` instead of `System.IOs`.

Do not use the same name for a namespace and a type in that namespace, e.g. do not use `CompanyX.Tools.Extensions` as a namespace name and then also provide a class named `Extensions` in the same namespace.

4. Names of Classes, Structs and Interfaces

Name classes and structs with nouns or noun phrases, using PascalCase. This distinguishes type names from methods, which are named with verb phrases.

Name interfaces with adjective phrases, or occasionally with nouns or noun phrases. Nouns and noun phrases should be avoided because they might indicate that the type should be an abstract class, and not an interface.

Do not give class names a prefix (e.g. “Cls” or “C”).

Consider ending the name of derived classes with the name of the base class. Doing so clearly describes the relationship. Some examples of this in code are: `ArgumentOutOfRangeException`, which is a kind of `Exception`, and `SerializableAttribute`, which is a kind of `Attribute`.

Prefix interfaces with the letter `I`, to indicate that the type is an interface. For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective) are appropriate interface names. As with other type names, avoid abbreviations.

Ensure that the names differ only by the `I` prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.

4.1 Names of Generic Type Parameters

Generics were added to .NET Framework 2.0. The feature introduced a new kind of identifier called type parameter.

Name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value. You should consider using `T` as the type parameter name for types with one single-letter type parameter.

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

Use prefix descriptive type parameter names with `T`

```
public interface ISessionChannel<TSession> where TSession : ISession
{
       TSession Session { get; }
}
```

5. Names of Type Members

5.1 Names of Methods

Methods are the means of taking action, so method names should be verbs or verb phrases. This convention serves to distinguish method names from property and type names, which are noun or adjective phrases.

Give methods names that are verbs or verb phrases e.g.:

```
public class String
{
        public int CompareTo(...);
        public string[] Split(...);
        public string Trim();
}
```

5.2 Names of Properties

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

Don't have properties that match the name of "Get" methods as in the following example:

```
public string TextWriter { get {...} set {...} }
public string GetTextWriter(int value) { ... }
```

The pattern above typically suggests that the property should be a method.

Name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection" e.g. call your collection of colours, `Colours` rather than `ColourList` or `ColourCollection`.

Name *boolean* properties with an affirmative phrase (`CanSeek` instead of `CantSeek`). Optionally, you can also prefix *boolean* properties with "Is," "Can," or "Has," but only where it adds value.

Consider giving a property the same name as its type.

```
public enum Colour {...}
public class Control
{
        public Colour Colour { get {...} set {...} }
```

```
}
```

5.3 Names of Events

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised e.g.:

- `Clicked`
- `Painting`
- `DroppedDown`
- `WillAppear`

Give events names with a concept of before and after, using the present and past tenses. Don't use "Before" or "After" postfixes to indicate pre- and post-events. Use present and past tenses as just described.

Name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

```
public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

Use two parameters named sender and e in event handlers. The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

Name event argument classes with the "EventArgs" suffix.

5.4 Names of Fields

Use PascalCase for field names. Name fields using a noun phrase or adjective. Do not use a prefix on a field name.

6. Names of Parameters

Use camelCasing in parameter names. Use descriptive parameter names; consider using names based on a parameter's meaning rather than its type.

7. Names of Unit Test Methods

When a test fails there are 3 questions that are pertinent to speedy resolution of the unit test:

- What was being tested?
- Under which circumstances?
- What was the expected result?

These questions define our naming convention represented by the following template:

`<UnitOfWork>_<StateUnderTest>_<ExpectedBehaviour>`

`UnitOfWork` describes the method being tested. `StateUnderTest` describes the state and conditions for which you are testing. Finally, `ExpectedBehaviour` describes the expected outcome that is being sought by the unit test. Below are some examples:

- `IsAdult_AgeLessThan18_False`
- `WithdrawMoney_InvalidAccount_ExceptionThrown`
- `AdmitStudent_MissingMandatoryFields_FailToAdmit`

The AAA (arrange, act and assert) pattern should be used to form the backbone of your unit test.

- The Arrange section of a unit test method initialises objects and sets the value of the data that is passed to the method under test.
- The Act section invokes the method under test with the arranged parameters.
- The Assert section verifies that the action of the method under test behaves as expected.

Here is a real world example of the pattern in practice:

```
public void Credit_WithValidAmount_UpdatesBalance()
{
        // arrange
        double beginningBalance = 10.00;
        double creditAmount = 5.00;
        double expected = 15.00;
        BankAccount account = new BankAccount("Mr. Darth Vader", beginningBalance);

        // act
        account.Credit(creditAmount);

        // assert
        double actual = account.Balance;
        Assert.AreEqual(expected, actual, 0.001, "Account not credited correctly");
}
```

8. Async/Await Patterns

This section is a brief summary of an article by Microsoft MVP Stephen Cleary:

https://msdn.microsoft.com/en-us/magazine/jj991977.aspx

8.1 Naming Convention

Name `async` methods with the "Async" suffix.

8.1 Avoid async void

You should use

- `async Task`

over

- `async void`

`Async Task` methods enable easier error handling, composability and testability. The one exception to this rule is asynchronous event handlers.

8.2 Async all the way

You should avoid mixing asynchronous and blocking code because it can cause deadlocks, more complex error handling and unexpected blocking of context threads.

See below for preferred use of `async` features:

To do this …	Instead of this …	Use this …
Retrieve the result of a background task	`Task.Wait or Task.Result`	`await`
Wait for any task to complete	`Task.WaitAny`	`await Task.WhenAny`
Retrieve the results of multiple tasks	`Task.WaitAll`	`await Task.WhenAll`
Wait a period of time	`Thread.Sleep`	`await Task.Delay`

8.3 Configure Context

You should use `ConfigureAwait(false)` wherever possible. Context-free code has better performance for GUI applications and is a useful technique for avoiding deadlocks when working with a partially async codebase.

It's useful to think of `async/await` as using two separate threads to execute one method: one to begin it, and a different one to complete it. Using `ConfigureAwait(false)` is the simplest way to indicate to the compiler/runtime that it need not give the second thread the context that the first one had. It should be thought of as an optimisation hint; a good rule of thumb is that you should use `ConfigureAwait(false)` unless you know that you do need the context.

Here is an example of the two contrasting approaches:

```
private async Task HandleClickAsync()
{
	// can use ConfigureAwait here.
	await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
}

private async void button1_Click(object sender, EventArgs e)
{
	button1.Enabled = false;

	try
	{
		// can't use ConfigureAwait here.
		await HandleClickAsync();
	}
	finally
	{
		// we are back on the original context for this method.
		button1.Enabled = true;
	}
}
```

9. Exception Handling

Don’t hide exceptions with empty catches as is done here:

```
public void DivideByZero()
{
        try
        {
                var result = 10/0;
        }
        catch(Exception e)
        {
                // empty catch = ignored and invisible problem
        }
}
```

Don’t re-raise errors in a catch block in this way:

```
throw e;
```

This pattern will reset the stack trace and make it appear as though the error came from within the catch.

You should rather do this:

```
throw;
```

This pattern does not reset the stack trace and will preserve the original offending statement.

10. Other

10.1 Newlines/Whitespace

Consider keeping excessive use of newlines/whitespace to an absolute minimum.

10.2 Ternary Statements

Nested ternary statements must be appropriately indented so as to improve readability.

10.3 Superfluous Parameters

Please ensure that method signatures are free of unused parameters except in cases where they are required e.g. overrides and event handlers.

10.4 Superfluous Variables

Do not create *assigned-to-but-never-used-variables*. These variables are highlighted in Visual Studio as a warning during compile time; keep your warnings clean.

10.5 Class per File

You should only have one class per project file. Partial classes need to be placed in their own file but the file name must elaborate on the part of the class that the file contains as follows:

```
<OriginalClass>.<Part>.cs
```

`MyClass.cs`	[original class file]
`MyClass.PartTwo.cs`	[partial class file]

10.6 Floating-Point Comparisons

When comparing floating-point numbers be sure to use epsilon.

Here is an example of how not to do floating-point comparisons:

```
public bool CompareFloatingPointValuesPoorly()
{
	var aThirdOfTen = 10f / 3f;
	var aThirdOfTenTimesThree = aThirdOfTen * 3;

	return aThirdOfTen == aThirdOfTenTimesThree;
	// returns false :(
}
```

Here is an example of the correct pattern to follow:

```
public bool CompareFloatingPointValuesAppropriately()
{
	var aThirdOfTen = 10f / 3f;
	var aThirdOfTenTimesThree = aThirdOfTen * 3;

	return Math.Abs(10 - aThirdOfTenTimesThree) <= float.Epsilon;
	// returns true :)
}
```

10.7 String Comparison

Some Unicode characters have multiple equivalent binary representations consisting of sets of combining and/or composite Unicode characters. The nasty side-effect of this is that two strings can ***look*** the same but actually contain different characters.

The following pattern is hazardous:

```
public bool CompareStringsPoorly()
{
        var firstString = "naboo";
        var secondString = "naboo";

        return firstString == secondString;
}
```

This pattern is correct:

```
public bool CompareStringsAppropriately()
{
        var firstString = "death star";
        var secondString = "death star";

        return firstString.Equals(secondString, StringComparison.Ordinal);
}
```

The `StringComparison.Ordinal` parameter will assert that both strings are ***lexically*** equal.

Providing insight to the inner workings and history of Unicode is beyond the scope of this document. If you don't have a good understanding of Unicode (which you should) we *urge* you to read this timeless article titled "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" by Joel Spolsky.

10.8 Pseudo-code & Code Comments

If you have followed the standards and conventions in this document you will have written self-documenting and readable code, meaning that it doesn't require additional comments to be able to understand, read and navigate your way through a code base.

Be that as it may, there are occasions where commenting might be useful but you should use code comments sparingly and conservatively.

10.9 String Literals

Use `nameof()` rather than string literals where logic involves names of properties, parameters and variables.

Here is a typical use case for `nameof()`:

```
switch (e.PropertyName)
{
	// this way ...
	case nameof(SomeOtherPropertyName):
		break;

	// ... as opposed to this way
	case "YetAnotherPropertyName":
		break;
}
```

In the above example, problems associated with renaming the `SomeOtherPropertyName` will be taken care of by the compiler, whereas the second case will not, most probably creating an oversight that will lead to problems during runtime.

10.10 Code Regions

Code regions (`#region`) are widely considered to be ineffective, counter-productive and more often than not hinders readability. The temptation to use regions inside methods should be avoided. You should rather refactor your method.

10.11 Terse Getters & Setters

Keep your getters and setters terse. A getter and setter should preferably be one line of code. If complex logic is required encapsulate the login into a clad method that is descriptive and follows the rules set out in this document.

10.12 Expression Bodied Members

We encourage the use of expression bodied members (EBM), but please ensure that you format all instances consistently as is shown below.

Simple example:

```
protected override IMvxApplication CreateApp()
            => new App();
```

Notice how a new line is created (with indentation) and starts with the lambda (=>) operator.

Intricate example:

```
private MvxCommand _command;
public MvxCommand Command
       => _command ?? (_command = new MvxCommand(NavigateToBilling));

private void NavigateToBilling()
{
       if (IsOwnNetwork)
       {
              ShowViewModel<MyBillProductWalletViewModel>();
       }
       else
       {
              ShowViewModel<MyBillViewModel>();
       }
}
```