C# Programming Recommendations

Method / User Guide

Revision: 6.1

Validity Date: 2019/11/05

C# Programming Recommendations

Coding Style Guideline for C#

Naming

Naming rules

- Do not create identifiers which differ in letter capitalization only. This will lead to problems in inter-operability scenarios.
- Do not use abbreviations for any identifier. For instance use "Windows" instead of "Win" or "Length" instead of "Len".
- Use an acronym only when it is well known. (As a rule of thumb: Acronyms which wikipedia does not directly know for the entity they are supposed to describe are not considered as well known.)
- Do not use an acronym only because it was popular in the past. For instance do not suffix identifiers with "ex". Use meaningful phrases instead.
- For acronyms the same rules as for any other word applies.
- At the beginning of a camelCase acronyms are not capitalized.

Positive acronym examples are:

Acronym	Reason
Io	Accepted for "Input Output" (http://en.wikipedia.org/wiki/Input/output)
Html	Accepted for "Hypertext Markup Language" (http://en.wikipedia.org/wiki/Html)
Dicom	Accepted for "Digital Imaging and Communications in Medicine" (http://en.wikipedia.org/wiki/Dicom)
Id	Accepted for "Identifier" (http://en.wikipedia.org/wiki/Id)

More positive examples are: Ok, Xml

Negative acronym examples are:

Bad acronym	Reason
Fact	Unknown acronym for "factory"
Svc	Neither a search engine nor wikipedia resolve this to "Service"
MMF	According to http://de.wikipedia.org/wiki/MMF this acronym means "make money fast" instead of "memory mapped file"
DEX	http://en.wikipedia.org/wiki/DEX resolves to "Dicționarul explicativ al limbii române"

More negative acronym examples are: Impl, Cfg, Mgr, Svr, Exec, MEx, Btn, MSMQ, Ctx, MER, GOM, sys, AOM, Cmd, STR

Capitalization examples:

camelCase	PascalCase	Wrong
fileName	FileName	filename, Filename
logOn	LogOn	Login, Logon
callback	Callback	cb, CallBack
userCredentials	UserCredentials	Usercredentials
controlCommand	ControlCommand	ctrlcmd, Controlcommand
communicationContext	CommunicationContext	commuCtx, commuContext

Following table displays naming rules applying for C# language constructs. It must be ensured that within a class a consistent style is used. Do not mix allowed variations within one class. A dash in the prefix or suffix column indicates that the artifact must not have a prefix or suffix. Examples of identifiers that are not allowed: aTimeout, theList, vector_in. Ideally the name of an identifier is close to the name of its type: TimeSpan myTimeSpan, CacheMetaData cacheMetaData and so on.

	Identifier			
	Prefix	**Body**$^{1)}$	**Suffix**	**Example**
File Naming	-	- Matches exactly the single public type in the file	-	NetworkManager.cs
Class type	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	-	class NetworkManager
Struct type	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	-	struct NetworkManager
Interface	I	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	-	interface IComparable
Enumeration	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase (recommended: in singular)	-	enum ClassOperationMode

Enumeration item	-	- Pascal case
		- No abbreviation
		- No underscore
		- Noun or noun phrase in singular
			-	enum ClassOperationMode
				{
				OperationMode1,
				OperationMode2
				}

Method	-	- Pascal case - Verb oder verb phrases	-	void DoCommand();
Method parameters	-	- Camel case - No abbreviation - No underscore	-	void Save(String fileName);

Type parameters	T	- Pascal case
		- No abbreviation
		- No underscore
		- Noun or noun phrase
				class Stack<TItem> { ... }
				TData GetData<TData>(String id);

Method local variables	-	- Camel case - No abbreviation - No underscore	-	int localCounter;
Field	my	- Camel case - No abbreviation - No underscore	-	private String myPeerName;
Static field	my	- Camel case - No abbreviation - No underscore	-	private static String myNodeName;
Const field	-	- Pascal case - Noun or noun phrases - No abbreviation - No underscore	-	private const String NodeName = String.Empty;
Property	-	- Pascal case - Noun or noun phrases - No abbreviation - No underscore	-	public String PeerName {get; set;}
Exception class	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	Exception	SyngoException ArgumentOutOfRangeException
Namespaces	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase - Exceptions for: brand names	-	syngo.Common.Core
Delegate	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	Handler or Callback	delegate void LogHandler(String msg);
Event	-	- Pascal case - No abbreviation - No underscore - Noun or noun phrase	-	event LogHandler LogMessage;

1) Please find definitions in chapter General Programming Recommendations - Naming Basics.

Layout

Definitions

Term	Definition	Example

Indentation	Indentation of blocks by whitespaces	{
		    counter++;
		    Foo(counter);
		}

Statement	One single language instruction delimited by a semicolon.	counter++;

Statement block	Two or more statements that are sequentially executed.	counter++;
		Foo(counter);

Curly brackets	{ or } delimiting a single statement or statement block	{
		    counter++;
		    Foo(counter);
		}

Layout Rules

All code follows the so called "ANSI Indentation". This style puts the brace associated with a statement block on the next line, indented to the same level as the statement block. Statements within the braces are indented to the next level. One indentation must be 4 whitespaces and no tab control character.

Layouting with curly brackets:

Layout rule	Positive example	Negative example

Single statements or statement blocks are always surrounded by curly brackets	if (doesApply)
	{
	    counter++;
	}
		if (doesApply)
		counter++;

If a statement spans multiple lines, the opening curly bracket must be placed on its own line.	if (doesApply)
	{
	    counter++;
	}
	
	public int Counter
	{
	    get; set;
	}
		if (doesApply) {
		    counter++;
		}
		
		public int Counter {
		    get;
		}

A statement containing curly brackets must not be placed on a single line. The opening and closing curly brackets must each be placed on their own line.	if (doesApply)
	{
	    counter++;
	}
		if (doesApply) {
		counter++; }

An accessor can only be placed on a single line if all of the accessors in the property are placed on a single line.	public int Counter
	{
	    get { return counter; }
	    set { counter = value; }
	}
		public int Counter
		{
		    get
		    {
		        return counter;
		    }
		    set { counter = value; }
		}

Layouting with line spacing:

Layout rule	Positive example	Negative example
There is never more than one blank line in a row	... } public Foo() { ...	... } public Foo() { ...
An opening curly bracket must not be preceded or followed by a blank line	if (doesApply) { counter = 1; }	if (doesApply) { counter = 1; if (doesApply) { counter = 1;
A closing curly bracket must not be preceded by a blank line	if (doesApply) { counter = 1; }	if (doesApply) { counter = 1; }
A closing curly bracket must be followed by a blank line or another closing curly bracket.	... } public Foo() { ...	... } public Foo() { ...

Layouting with spacing:

Layout rule	Positive example	Negative example
Binary operators must be surrounded by blanks	1 + 1	1+ 1
Unary operators must not be surrounded by blanks	++i;	++ i;
A comma must be followed by a blank or a carriage return	{"a", "b", "c"}	{"a","b" ,"c"}
A semicolon must not be preceded by a blank or a blank line	i++;	i++ ;
A semicolon must be followed by a blank or a carriage return	for(var i = 0; i < 10; i++)	for(var i = 0;i< 10;i++)
Instead of the "Tab" special character 4 whitespaces are used		

Programming

For the rest of the document the terms "public", "protected", "private" and "internal" are always refering to the corresponding compiler access modifiers. Any other usage of those terms for comments or similiar is explicitly described as such.

Assemblies

- Do not split source code implementing one concern into several assemblies. (PA1)

- Do not split the implementation of one component into several assemblies in order just to hide implementation portions of it. If that is necessary the underlying design must be reviewed. Assemblies containing interface definitions only are allowed. (PA2)

Source files

- A source file contains exactly one public or one internal class, struct or interface. Generic versions of one entity are allowed in the same file. (PS1)
- A source file can contain an arbitrary amount of private classes, structs or interfaces. (PS2)
- A source file has exactly the name of the one public or internal class, struct or interface it contains. (PS3)
- A source file contains entities for exactly one namespace. (PS4)
- A source file is located in a folder hierachy relatively to its project root matching the single namespace it contains. (PS5)
- Each filename including its path and extension must not exceed a maximum length of 256 characters. (PS7)
- Paths accessible via the syngoBASE directory must not exceed 90 characters in total including the drive letter, colon, backslash, syngoBASE prefix and the file name. (PS8)
- Classes must not be split over several source files by using the partial keyword unless the class contains generated code. (PS9)

Interfaces

- Interfaces cover a single type of usage (role specific). If there is a "management view" and an "operational view", use two interfaces. (PI1)
- Interfaces are kept slim. (PI2)
- Interfaces do not provide overloads. (PI3)
- Interfaces are orthogonal to other interfaces. That means that interfaces never cover functionality other interfaces already cover. If so the underlying design must be reviewed. (PI4)
- Interfaces are never casted to classes. If that is necessary the underlying design must be reviewed. (PI7)
- Do not create interfaces which are used to tag or put an attribute on an object. If that is necessary other language constructs like attributes should be used. (PI8)
- Interfaces are never empty. (PI9)
- If an interface is not implemented by other users consider using an abstract class or normal class. (PI10)
- Give hints in your documentation where implementations of an interface can be found. (PI11)

Abstract classes

- Abstract classes must not have a public constructor. (PAC1)
- With every abstract class at least one concret derived type should be available. (PAC2)
- Give hints in the documentation where implementations of an abstract class can be found. (PAC3)
- Abstract classes define semantics for concret derived types. (PAC4)

Classes

- A class should not implement more than one responsibility. (PC1)
- A class which is used by exactly one other class should be nested in this class. (PC2)
- The visiblity of a class should be restricted as much as possible. (PC3)

Methods

- The body of a method should not exceed 100 statements. (PM1)
- Methods should not have more than 3 parameters. If more data is necessary a wrapper class should be used as parameter. (PM2)
- Static members should be accessed by the hosting class or struct name. (PM4)
- Conditional nesting level (if, for, do, while, switch) should not exceed a depth of 3. If that is necessary the underlying design must be reviewed. (PM5)
- Methods which main purpose is to set or get data must be prefixed with "set" and "get" respectively. (PM7)

- Public and protected methods must verify parameters for validity. (PM8)
- Do not repeat yourself (DRY). Code that appears more than once must be refactored out into a (private) method. (PM9)
- The visiblity of a method should be restricted as much as possible. (PM10)
- Methods implementing an interface method or overriding a virtual method must provide exactly the same signature including identifiers. (PM11)
- A method should be made static when possible. (PM12)
- For parameters with a default value the default value must be null (or default(T)). (PM14)

Properties

- Properties must get or set states in their hosting instance / type only. (PP1)
- Properties must not call methods in other types than the one they are hosted in. (PP2)
- Properties must not hide data retrieval from other sources than the hosting instance / type. If that is necessary a method must be made available. (PP3)
- Properties must not be used in interfaces or classes which are meant to be used as remoting proxies or similar. (PP4)
- Property getters must not throw any exceptions. (PP5)
- Property getters must not modify the state of the object or class. (PP6)

Events

- An event raising method must be prefixed with "Raise". It must not be prefixed with "Trigger" or "Fire". (PE2)

Parameters

- Do not pass boolean values as parameters but use enumerations instead when writing a public API. (PPR1)
- Avoid "out" parameters. Pack the additional "out data" in a wrapper classes for the method result instead. (PPR2)
- Consider using "out" parameters for performance reasons in highly frequently executed code. (PPR3)

Variables and Fields

- Variables must be declared within the smallest possible scope. (PVF1)
- Variables and Fields should be initialized with a value on declaration only if necessary. (PVF2)
- Fields must be always marked as private. Exceptions can be low-level performance optimizations. (PVF3)

Enumerations

- Enumerations must not have values which are "reservered for future use". (PEN1)
- Enumerations always have 2 or more values. (PEN2)
- Members of a normal enumeration (which does not list flags) must not have any assigned values. (PEN3)

API Documentation

- All public interfaces, classes and structs definitions must be documented with a XML documentation header. (PD1)
- All public or internal methods, properties and event signatures must be documented with a XML documentation header. (PD2)
- All protected and private method, property and event signatures can be considered being documented with a XML documentation header. (PD3)

If method is documented:

- Method documentation must contain purpose of the method, required preconditions, meaning of each parameter, resulting postconditions, meaning and possible content of the return parameter. (PD4)
- Method documentation must contain list of exceptions that can be thrown directly by the method. (PD5)
- All documentation must be written in precise english. Rule of thumb: Another developer should be able to provide same semantic behavior just by reading its documentation. (PD7)
- If method returns a success/error code its documentation must contain a description of the possible error codes. (PD8)

Localization

- Do not hardcode user interface resources. (LO1)
- Do not use localized strings for comparison operations. (LO2)
- Store strings, when possible, in the invariant culture format. This makes it easier within the application to store and restore them. Furthermore the application has a dedicated base culture to make conversion to or from a specific culture. (LO3)
- Do not use composite strings that are built at run time from concatenated phrases. Composite strings are difficult to localize because they often assume an English grammatical order that does not apply to all languages. (LO4)
- For string handling always use the .NET string classes. (LO5)
- If you make a string comparison within your application for internal program flow always use the culture invariant comparison. (LO6)

Globalization

To handle globalization unicode plays the key role.

- Use the culture-aware classes provided by the System.Globalization namespace to manipulate and format data. (GLO2)
- Use the SortKey and CompareInfo class for sorting. (GLO3)
- Use the CompareInfo class for string comparisons. (GLO4)
- Use the DateTimeFormatInfo class for date and time formatting. (GLO5)
- Use the NumberFormatInfo class for numeric formatting. (GLO6)
- Use the Calendar class or one of the specific Calendar implementations for Gregorian and non-Gregorian calendars. (GLO7)
- Use the culture property settings provided by the System.Globalization.CultureInfo class in the appropriate situations. Use the CultureInfo.CurrentCulture property for formatting tasks, such as date and time or numeric formatting. Use the CultureInfo.CurrentUICulture property to retrieve resources. (GLO8)
- Enable your application to read and write data to and from a variety of encodings by using the encoding classes in the System.Text namespace. Do not assume ASCII data. Assume that international characters will be supplied anywhere a user can enter text. For example, accept international characters in server names, directories, file names, user names, and URLs. (GLO9)
- When using the UTF8Encoding class for security reasons it is recommended that you use the error detection feature offered by this class. To turn on the error detection feature, create an instance of the class using the constructor that takes a throwOnInvalidBytes parameter and set the value to true. (GLO10)
- Whenever possible, handle strings as entire strings instead of as a series of individual characters. This is especially important when sorting or searching for substrings. This will prevent problems associated with parsing combined characters. (GLO11)
- When using WinForms: Display text using the classes provided by the System.Drawing namespace. (GLO12)
- For consistency across operating systems, do not allow user settings to override CultureInfo. Use the CultureInfo constructor that accepts a useUserOverride parameter and set it to false. (GLO13)
- If a security decision is based on the result of a string comparison or case change operation, perform a culture-insensitive operation by explicitly specifying the CultureInfo.InvariantCulture property. This practice ensures that the result is not affected by the value of CultureInfo.CurrentCulture. (GLO15)
- When dealing with strings to figure out how many text elements are contained within a string use class System. Globalization.StringInfo. The .NET Framework defines a text element as a unit of text that is displayed as a single character, i.e., a grapheme. A text element can be a base character, a surrogate pair, or a combining character sequence. The Unicode Standard defines a surrogate pair as a coded character representation for a single abstract character that consists of a sequence of two code units, where the first unit of the pair is a high-surrogate and the second is a low-surrogate. The Unicode Standard defines a combining character sequence as a combination of a base character and one or more combining characters. A surrogate pair can represent a base character or a combining character. For more information on surrogate pairs and combining character sequences see The Unicode Standard at http://www.unicode.org/. (GLO16)

Return values and Exceptions

To validate input parameters and return values code contracts must be used where available. Following applies only for public and internal legacy methods or methods where code contracts are **not** available:

- When an input parameter is null where null is not allowed as value an "ArgumentNullException" must be thrown. The Exception must contain the name of the parameter. (RE1)
- When an input parameter is outside the allowable range of values as defined by the invoked method an "ArgumentOutOfRangeException" must be thrown.The Exception must contain the name of the parameter. (RE2)
- When an input parameter is invalid in general terms an "ArgumentException" must be thrown. The Exception must contain the name of the parameter. (RE3)

Multithreading

- Avoid nesting lock statements. (MT1)
- Do not call code from another class from within a lock. (MT2)
- Do not call delegates or events from within a lock. (MT3)
- Do not expose lock objects. (MT4)

Performance

- Always use the generic version of a collection. (PF2)
- Use String.IsNullOrEmpty to check for null or empty strings. (PF3)

General

- Use inclusive lower limits and exclusive upper limits in loop constructs. (PG2)
- Avoid using goto. The keywords break and continue are accepted. (PG4)
- A switch instruction always contains a default branch. The default branch must not be empty. In that case it must throw an exception. (PG5)
- Use parentheses to clarify the order of evaluation for operators in expressions. (PG6)
- All code must compile without any warnings at the highest compiler warning level except for warnings excluded by [4]. (PG8)
- All language constructs which should be "private" must be marked with the access qualifier "private". (PG9)
- Always use positive logic. Do not name fields like for an example: doesNotApply, didNotHappenAmount and so forth. (PG10)
- Reference types like Strings, Collections or Arrays which are supposed to be empty must not point to null but an appropriate empty instance like for an example String.Empty or an empty List. (PG11)
- Classes which are serializable must not contain auto properties. (PG12)
- Consider using var to declare and assign local variables where the type of the right-hand side expression is obvious. (PG13)
- Avoid using var when the resulting type of the local variable is not easy to discern and it is important for the reader of the code to know the type of the variable. (PG14)

Patterns

Try Pattern

Consider providing methods which can throw exceptions as part of their regular execution - because the error is part of the environment (file is missing, mutex is locked etc) - also as a variation in the so called "try style". To transform a method into the try style the return value becomes an out parameter and as new return value a boolean is introduced which indicates success or failure of the call. A good example for the try style is the Dictionary<T>.TryGetValue method.

back to top

Rules for Safety Relevant Code

The rules as described in the PLM M 41 main method apply.

back to top

History

See the history in the PLM M 41 main method.

back to top

Supported Deliverables	Supported Activities
None	None
Explained by training	**Supported Tools**
None	None