**C# Coding Standards**

Your Company Name Here

Abstract

This document contains coding conventions and style guidelines that will ensure that C# code will be of consistent and superior quality. It includes general coding guidelines as well as discussions on naming, formatting, and language usage patterns.

Contents

[1. Introduction 3](#_Toc217699211)

[1.1 Purpose 3](#_Toc217699212)

[1.2 Scope 3](#_Toc217699213)

[1.3 Document Conventions 3](#_Toc217699214)

[1.4 Feedback 3](#_Toc217699215)

[2. C# Golden Rules 4](#_Toc217699216)

[3. Formatting 5](#_Toc217699217)

[3.1 Class Layout 5](#_Toc217699218)

[3.2 Indicating Scope 5](#_Toc217699219)

[3.3 Indentation & Braces 6](#_Toc217699220)

[3.4 White space 6](#_Toc217699221)

[3.5 Long lines of code 7](#_Toc217699222)

[4. Commenting 8](#_Toc217699223)

[4.1 Intellisense Comments 8](#_Toc217699224)

[4.2 End-Of-Line Comments 8](#_Toc217699225)

[4.3 Single Line Comments 8](#_Toc217699226)

[4.4 // TODO: Comments 8](#_Toc217699227)

[4.5 C-Style Comments 8](#_Toc217699228)

[5. Capitalization & Naming 9](#_Toc217699229)

[5.1 Capitalization 9](#_Toc217699230)

[5.2 Naming 9](#_Toc217699231)

[6. Programming 10](#_Toc217699232)

[6.1 Namespaces 10](#_Toc217699233)

[6.2 Classes & Structures 10](#_Toc217699234)

[6.3 Interfaces 10](#_Toc217699235)

[6.4 Constants 11](#_Toc217699236)

[6.5 Enumerations 11](#_Toc217699237)

[6.6 Variables, Fields & Parameters 11](#_Toc217699238)

[6.7 Properties 12](#_Toc217699239)

[6.8 Methods 13](#_Toc217699240)

[6.9 Event Handlers 13](#_Toc217699241)

[6.10 Error Handling 13](#_Toc217699242)

[Appendix A. Naming Parts & Pairs 15](#_Toc217699243)

[A.1 Common Adjective Pairs A.2 Common Property Prefixes 15](#_Toc217699244)

[A.3 Common Verb Pairs 15](#_Toc217699245)

[A.4 Common Qualifiers Suffixes 15](#_Toc217699246)

[Appendix B. References 16](#_Toc217699247)

[Revision History 17](#_Toc217699248)

# 1. Введение

В документе приводятся рекомендации по оформлению кода на языке C#. Нередко название подобных документов использует слово «стандарт». Автор считает что «рекомендация» является более подходящим словом так как термин стандарт подразумевает нечто сильно формализованное и практически неизменное. Стиль оформления кода является вопросом вкуса сформированным на основе опыта. Поэтому стили принятные на разных проектах могут сильно отличаться. Здравый смысл и командный дух важнее чем правила, которые можно изменить.

*Не важно какой именно стиль оформления кода вы применяете в своих проектах. Важно чтобы он был задокументирован и чтобы все члены команды следовали ему.*

В качестве основы для создания данного документа послужил файл «CSharp Coding Standards.doc» из статьи [Holiday Goodie Bag: Free C# and VB Coding Standards Reference Documents](http://www.notsotrivial.net/blog/post/2008/12/Holiday-Goodie-Bag-Free-C-and-VB-Coding-Standards-Reference-Documents.aspx).

## 1.1 Purpose

The purpose of this document is to provide coding style standards for the development of source code written in C#. Adhering to a coding style standard is an industry proven best-practice for making team development more efficient and application maintenance more cost-effective. While not comprehensive, these guidelines represent the minimum level of standardization expected in the source code of projects written in C#.

## 1.2 Scope

This document provides guidance on the formatting, commenting, naming, and programming style of C# source code and is applicable to component libraries. web services, web sites, and rich client applications.

## 1.3 Document Conventions

Example code is shown using the Code font and shows syntax as it would be color coded in Visual Studio’s code editor.

## 1.4 Обратная связь

Feedback on these guidelines is highly encouraged. Please send any questions or comments to your application architect.

# 2. «Золотые» правила C#

Нижеследующие рекомендации являются универсальными The following guidelines are applicable to all aspects C# development:

* Следуйте правилам оформления ранее написанного кода. Стремитесь к сохранению единого стиля в одном приложении. Использование разных стилей в одном проекте затрудняет командную работу!
* Make code as simple and readable as possible. Assume that someone else will be reading your code.
* Prefer small cohesive classes and methods to large monolithic ones.
* Use a separate file for each class, struct, interface, enumeration, and delegate with the exception of those nested within another class.
* Write the comments first. When writing a new method, write the comments for each step the method will perform before coding a single statement. These comments will become the headings for each block of code that gets implemented.
* Use liberal, meaningful comments within each class, method, and block of code to document the **purpose** of the code.
* Отмечайте незаконченный код с помощью // TODO: комментариев так как при одновременной работе над большим количеством классов легко потерять нить размышлений.
* Never hard code “magic” values into code (strings or numbers). Instead, define constants, static read-only variables, and enumerations or read the values from configuration or resource files.
* Prefer while and foreach over other available looping constructs when applicable. They are logically simpler and easier to code and debug.
* Use the StringBuilder class and it’s Append(), AppendFormat(), and ToString() methods instead of the string concatenation operator (+=) for much more efficient use of memory.
* Be sure Dispose() gets called on IDisposable objects that you create locally within a method. This is most commonly done in the finally clause of a try block. It’s done automatically when a using statement is used.
* Never present debug information to yourself or the end user via the UI (e.g. MessageBox). Use tracing and logging facilities to output debug information.
* Gaps and exceptions to these guidelines should be discussed and resolved with your application architect.

# 3. Formatting

## 3.1 Class Layout

Classes should be organized into regions within an application using a layout determined by your application architect. These may be based on accessibility, type, or functionality. Consult your architect for the layout strategy used in your application.

Example:

// Class layout based on accessibility

class Purchasing

{

#region Main

#region Public

#region Internal

#region Protected

#region Private

#region Extern

#region Designer Generated Code

}

Guidelines:

* Use the same layout consistently in all classes in an application.
* Omit regions if their associated class elements are not needed.
* The Designer Generated Code region created by Visual Studio’s Visual Designer should never be modified by hand. It should contain only code generated by the designer.

## 3.2 Indicating Scope

Indicate scope when accessing all static and non-static class members. This provides a crystal clear indication of the intended use of the member. VisualStudio.NET intellisense is automatically invoked when using this practice, providing a list of all available class members. This helps prevent unnecessary typing and reduces the risk of typographic errors.

Example:

string connectionString = DataAccess.DefaultConnectionString;

float amount = this.CurrentAmount;

this.discountedAmount = this.CalculateDiscountedAmount( amount, this.PurchaseMethod );

Guidelines:

* Include the this keyword before all member fields, properties and methods.
* Include the name of the class before all static fields, constants, fields, and methods.

## 3.3 Indentation & Braces

Statements should be indented (using tabs) into blocks that show relative scope of execution. A consistent tab size should be used for all indentation in an application. Braces, when necessary, should be placed directly below and aligned with the statement that begins a new scope of execution. Visual Studio.NET includes a keyboard short-cut that will automatically apply this format to a selected block of code.

Example:

float CalculateDiscountedAmount( float amount, PurchaseMethod purchaseMethod )

{

// Calculate the discount based on the purchase method

float discount = 0.0f;

switch( purchaseMethod )

{

case PurchaseMethod.Cash:

// Calculate the cash discount

discount = this.CalculateCashDiscount( amount );

Trace.Writeline( “Cash discount of {0} applied.”, discount );

break;

case PurchaseMethod.CreditCard:

// Calculate the credit card discount

discount = this.CalculateCreditCardDiscount( amount );

Trace.WriteLine( “Credit card discount of {0} applied.”, discount );

break;

default:

// No discount applied for other purchase methods

Trace.WriteLine( “No discount applied.” );

break;

}

// Compute the discounted amount, making sure not to give money away

float discountedAmount = amount – discount;

if( discountedAmount < 0.0f )

{

discountedAmount = 0.0f;

}

LogManager.Publish( discountedAmount.ToString() );

// Return the discounted amount

return discountedAmount;

}

## 3.4 White space

Liberal use of white space is highly encouraged. This provides enhanced readability and is extremely helpful during debugging and code reviews. The indentation example above shows an example of the appropriate level of white space.

Guidelines:

* Blank lines should be used to separate logical blocks of code in much the way a writer separates prose using headings and paragraphs. Note the clean separation between logical sections in the previous code example via the leading comments and the blank lines immediately following.
* Single spaces should be used to separate logical elements within individual statements. This can be seen clearly in the CalculateDiscountedAmount and switch statements in the preceding example. Note the spaces immediately after opening ‘(‘s and before closing ‘)’s.

## 3.5 Long lines of code

Comments and statements that extend beyond 80 columns in a single line can be broken up and indented for readability. Care should be taken to ensure readability and proper representation of the scope of the information in the broken lines. When passing large numbers of parameters, it is acceptable to group related parameters on the same line.

Example:

string Win32FunctionWrapper(

int arg1,

string arg2,

bool arg3 )

{

// Perform a PInvoke call to a win32 function,

// providing default values for obscure parameters,

// to hide the complexity from the caller

if( Win32.InternalSystemCall(

null,

arg1, arg2,

Win32.GlobalExceptionHandler,

0, arg3,

null )

{

return “Win32 system call succeeded.”;

}

else

{

return “Win32 system call failed.”;

}

}

Guidelines:

* When breaking parameter lists into multiple lines, indent each additional line one tab further than the starting line that is being continued.
* Group similar parameters on the same line when appropriate.
* When breaking comments into multiple lines, match the indentation level of the code that is being commented upon.
* Consider embedding large string constants in resources and retrieving them dynamically using the .NET ResourceManager class.

# 4. Комментарии

## 4.1 XML комментарии и Intellisense

Use triple slash ‘///’ comments for documenting the public interface of each class. This will allow Visual Studio.Net to pick up the method’s information for Intellisense. These comments are required before each public, internal, and protected class member and optional for private members.

## 4.2 Комментарии в конце строки

Use End-Of-Line comments only with variable and member field declarations. Use them to document the purpose of the variable being declared.

Example:

private string name = string.Empty; // Name of control (defaults to blank)

## 4.3 Single Line Comments

Use single line comments above each block of code relating to a particular task within a method that performs a significant operation or when a significant condition is reached. Comments should always begin with two slashes, followed by a space.

Example:

// Compute total price including all taxes

float stateSalesTax = this.CalculateStateSalesTax( amount, Customer.State );

float citySalesTax = this.CalculateCitySalesTax( amount, Customer.City );

float localSalesTax = this.CalculateLocalSalesTax( amount, Customer.Zipcode );

float totalPrice = amount + stateSalesTax + citySalesTax + localSalesTax;

Console.WriteLine( “Total Price: {0}”, totalPrice );

## 4.4 // TODO: комментарии

Используйте // TODO: комментарии чтобы отметить части кода которые надо написать, дописать или переписать до выпуска готовой версии. Source code should be searched for these comments before each release build.

## 4.5 C-Style Comments

Use c-style /\*…\*/ comments only for temporarily blocking out large sections of code during development and debugging. Code should not be checked in with these sections commented out. If the code is no longer necessary, delete it. Leverage your source control tools to view changes and deletions from previous versions of the code. If code must be checked in with large sections commented out, include a // TODO: comment above the block commented out describing why it was checked in that way.

# 5. Capitalization & Naming

## 5.1 Capitalization

Follow the standard [Naming Guidelines](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp) set by the .NET framework team by using only three capitalization styles: **Pascal**, **Camel**, and **Upper** casing.

Examples:

|  |  |  |
| --- | --- | --- |
| Identifier Type | Capitalization Style | Example(s) |
| Abbreviations | Upper | ID, REF |
| Namespaces | Pascal | AppDomain, System.IO |
| Classes & Structs | Pascal | AppView |
| Constants & Enums | Pascal | TextStyles |
| Interfaces | Pascal | IEditableObject |
| Enum values | Pascal | TextStyles.BoldText |
| Property | Pascal | BackColor |
| Variables, and Attributes | Pascal (public)  Camel (private, protected, local) | WindowSize  windowWidth, windowHeight |
| Methods | Pascal (public, private, protected)  Camel (parameters) | ToString()  SetFilter(string filterValue) |
| Local Variables | Camel | recordCount |

Guidelines:

* In **Pascal** casing, the first letter of an identifier is capitalized as well as the first letter of each concatenated word. This style is used for all public identifiers within a class library, including namespaces, classes and structures, properties, and methods.
* In **Camel** casing, the first letter of an identifier is lowercase but the first letter of each concatenated word is capitalized. This style is used for private and protected identifiers within the class library, parameters passed to methods, and local variables within a method.
* **Upper** casing is used only for abbreviated identifiers and acronyms of four letters or less.

## 5.2 Naming

Follow the standard set by the .NET framework team when it comes to naming. The **Programming** section of this document provides naming templates for each construct within the C# language. These templates can be used in conjunction with the tables provided in Appendix A. Naming Parts & Pairsto yield meaningful names in most scenarios.

# 6. Programming

## 6.1 Namespaces

Namespaces represent the logical packaging of component layers and subsystems. The declaration template for namespaces is: CompanyName.ProjectOrDomainName.PackageName.*SubsystemName*.

Examples:

Microsoft.Data.DataAccess

Microsoft.Logging.Listeners

Guidelines:

* Project and package level namespaces will normally be predetermined by an application architect for each project.
* Use Pascal casing when naming Subsystem namespaces.

## 6.2 Classes & Structures

Classes and structures represent the ‘Nouns’ of a system. As such, they should be declared using the following template: Noun + *Qualifier(s).* Classes and structures should declared with qualifiers that reflect their derivation from a base class whenever possible.

Examples:

CustomerForm : Form

CustomerCollection : CollectionBase

Guidelines:

* Use Pascal casing when naming classes and structures.
* Classes and structures should be broken up distinct #regions as previously described in the class layout guidelines.
* All public classes and their methods should be documented using the Intellisense triple slash ‘///’ comments built into Visual Studio.Net. Use this comment style to document the purpose of the class and its methods.
* Default values for fields should be assigned on the line where the field is declared. These values are assigned at runtime just before the constructor is called. This keeps code for default values in one place, especially when a class contains multiple constructors.

## 6.3 Interfaces

Interfaces express behavior contracts that derived classes must implement. Interface names should use Nouns, Noun Phrases, or Adjectives that clearly express the behavior that they declare.

Examples:

IComponent

IFormattable

ITaxableProduct

Guidelines:

* Prefix interface names with the letter ‘I’.
* Use Pascal casing when naming interfaces.

## 6.4 Constants

Constants and static read-only variables should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

public const int DefaultValue = 25;

public static readonly string DefaultDatabaseName = “Membership”;

Guidelines:

* Use Pascal casing when naming constants and static read only variables.
* Prefer the use of static readonly over const for public constants whenever possible. Constants declared using const are substituted into the code accessing them at compile time. Using static readonly variables ensures that constant values are accessed at runtime. This is safer and less prone to breakage, especially when accessing a constant value from a different assembly.

## 6.5 Enumerations

Enumerations should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Example:

/// <summary>

/// Enumerates the ways a customer may purchase goods.

/// </summary>

[Flags]

public enum PurchaseMethod

{

All = ~0,

None = 0,

Cash = 1,

Check = 2,

CreditCard = 4,

DebitCard = 8,

Voucher = 16,

}

Guidelines:

* Use Pascal casing when naming enumerations.
* Use the [Flags] attribute only to indicate that the enumeration can be treated as a bit field; that is, a set of flags.

## 6.6 Variables, Fields & Parameters

Variables, fields, and parameters should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

int lowestCommonDenominator = 10;

float firstRedBallPrice = 25.0f;

Guidelines:

* Use Camel casing when naming variables, fields, and parameters.
* Define variables as close as possible to the first line of code where they are used.
* Declare each variable and field on a separate line. This allows the use of End-Of-Line comments for documenting their purpose.
* Assign initial values whenever possible. The .NET runtime defaults all unassigned variables to 0 or null automatically, but assigning them proper values will alleviate unnecessary checks for proper assignment elsewhere in code.
* Avoid meaningless names like i, j, k, and temp. Take the time to describe what the object really is (e.g. use index instead of i; use swapInt instead of tempInt).
* Use a positive connotation for boolean variable names (e.g. isOpen as opposed to notOpen).

## 6.7 Properties

Properties should be declared using the following template: *Adjective(s)* + Noun + *Qualifier(s)*

Examples:

public TotalPrice

{

get

{

return this.totalPrice;

}

set

{

// Set value and fire changed event if new value is different

if( !object.Equals( value, this.totalPrice )

{

this.totalPrice = value;

this.OnTotalPriceChanged();

}

}

}

Guidelines:

* Use the common prefixes for inspection properties (properties that return query information about an object). See Appendix A. Naming Parts & Pairs for common prefixes.
* When there is a property setter that sets another property:
  + If the code in the other property sets a private member field in the same class, the field should be set directly, without calling the property setter for that field.
  + If a property setter sets a private field that would normally be set via another property setter, the originating setter is responsible for firing any events the other setter would normally fire (e.g. Changed events).
  + If a value that needs to be set that does NOT correspond to a private field, then an appropriate property setter or method should be called to set the value.

## 6.8 Methods

Methods should be named using the following format: Verb+ *Adjective(s)* + Noun + *Qualifier(s)*

Example:

private Ball FindRedCansByPrice(

float price,

ref int canListToPopulate,

out int numberOfCansFound )

Guidelines:

* Parameters should be grouped by their mutability (from least to most mutable) as shown in the example above.
* If at all possible, avoid exiting methods from their middles. A well written method should only exit from one point: at its end.
* Avoid large methods. As a method’s body approaches 20 to 30 lines of code, look for blocks that could be split into their own methods and possibly shared by other methods.
* If you find yourself using the same block of code more than once, it’s a good candidate for a separate method.
* Group like methods within a class together into a region and order them by frequency of use (i.e. more frequently called methods should be near the top of their regions.

## 6.9 Event Handlers

Event handlers should be declared using the following format: ObjectName\_EventName

Example:

private HelpButton\_Click( object sender, EventArgs e )

## 6.10 Error Handling

Use exceptions only for exceptional cases, not for routine program flow. Exceptions have significant performance overhead.

Guidelines:

* Pass a descriptive string into the constructor when throwing an exception.
* Use grammatically correct error messages, including ending punctuation. Each sentence in the description string of an exception should end in a period.
* If a property or method throws an exception in some cases, document this in the comments for the method. Include which exception is thrown and what causes it to be thrown.
  + Example: Comment for Order.TotalCost property might read "Gets or sets the total cost of an Order. If the TotalCost property is set when the cost should be calculated, an InvalidOperationException is thrown."
* Use the following exceptions if appropriate:
* ArgumentException (and ArgumentNull, ArgumentOutOfRange, IndexOutOfRange): Used when checking for valid input parameters to method.
* InvalidOperationException: Used when a method call is invalid for the current state of an object.

Example: TotalCost cannot be set if the cost should be calculated. If the property is set and it fails this rule, an InvalidOperationException is thrown.

* NotSupportedException: Used when a method call is invalid for the class.

Example: Quantity, a virtual read/write property, is overridden by a derived class. In the derived class, the property is read-only. If the property is set, a NotSupportedException is thrown.

* NotImplementedException: Used when a method is not implemented for the current class.

Example: A interface method is stubbed in and not yet implemented. This method should throw a NotImplementedException.

* Derive your own exception classes for a programmatic scenarios. All new derived exceptions should be based upon the core Exception class.

Example: DeletedByAnotherUserException : Exception. Thrown to indicate a record being modified has been deleted by another user.

* Rethrow caught exceptions correctly.

The following example throws an exception caught and rethrown incorrectly:

catch( Exception ex )  
{  
    LogManager.Publish( ex );  
    **throw ex; // INCORRECT – we lose the call stack of the exception**}

We log all unhandled exceptions in our applications, but may sometimes throw them again to let the higher level systems determine how to proceed. The problem comes in with the throw – it works much better to do this:

catch( Exception ex )  
{  
    LogManager.Publish( ex );  
**throw;   // CORRECT - rethrows the exception we just caught**}

Notice the absence of an argument to the throw statement in the second variation.

The difference between these two variations is subtle but important. With the first example, the higher level caller isn’t going to get all the information about the original error. The call stack in the exception is replaced with a new call stack that originates at the “**throw ex**” statement – which is not what we want to record. The second example is the only one that actually re-throws the original exception, preserving the stack trace where the original error occurred.

# 7. Полезные программы

## 7.1 JetBrains ReSharper

TODO

## 7.2 CodeStyle

TODO

# Приложение A. Части имен и пары

## A.1 Общепринятые пары прилагательных A.2 Общепринятые префиксы для свойств

|  |  |  |
| --- | --- | --- |
| Old…/New… |  | Allow… (Allows…) |
| Source…/Destination… |  | Can… |
| Source…/Target… |  | Contains… |
| First…/Next…/Current…/Previous…/Last… |  | Has… |
| Min…/Max… |  | Is… |
|  |  | Use… (Uses…) |

## A.3 Общепринятые пары глаголов

|  |  |
| --- | --- |
| Add…/Remove… | Open…/Close… |
| Insert…/Delete… | Create…/Destroy… |
| Increment/…Decrement… | Acquire…/Release… |
| Lock…/Unlock… | Up…/Down… |
| Begin…/End… | Show…/Hide… |
| Fetch…/Store… | Start…/Stop… |
| To…/From… (*Convert* implied) |  |

## A.4 Common Qualifiers Suffixes

|  |  |
| --- | --- |
| …Avg | …Limit |
| …Count | …Ref |
| …Entry | …Sum |
| …Index | …Total |

Note: Avoid using **Num** because of semantics; use **Index** and **Count** instead. Also, avoid using **Temp**; take the time to describe what the object really is (e.g. use **SwapValue** instead of **TempValue**).

# Appendix B. References

The following references were used to develop the guidelines described in this document:

* [*.Net Framework General Reference: Design Guidelines for Class Library Developers*](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp) – MSDN Online Reference
* *Code Complete* - McConnell
* *Writing Solid Code* - Macguire
* *Practical Standards for Microsoft Visual Basic* - Foxall
* *The Elements of Java Style* – Vermeulen, et. al.
* *The Elements of C++ Style* – Misfeldt, et. al.