NetRom

**Standard**

.NET Code Conventions

Version

12-Oct-2013



**.NET Code Conventions**

Standard

Caracal Blvd 178

200746, Craiova, Dolj

Romania

tel.: +40 251 437 738

fax.: +40 251 436 025

[soft@netrom.ro](mailto:soft@netrom.ro)

[www.netrom.ro](http://www.netrom.ro)

**Revision history**

|  |  |
| --- | --- |
| Title | .NET Code Conventions |
| Version | 2.0 Final |
| Subject | Standard |
| Author | NetRom |

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Author | Comments |
| 08-Jul-2007 | 0.1 | NetRom | Initial draft |
| 10-Dec-2013 | 1.0 | NetRom | Aligned to latest document template. |
| 05-Jun-2015 | 2.0 | NetRom | Updated dates format |

# Contents

[1. Contents 4](#_Toc374431923)

[2. About this document 5](#_Toc374431924)

[3. Files 6](#_Toc374431925)

[3.1. Project files, Solution files and namespaces 6](#_Toc374431926)

[3.2. Class files 6](#_Toc374431927)

[4. Indentation 7](#_Toc374431928)

[4.1. Wrap long lines 7](#_Toc374431929)

[4.2. Indentation characters 7](#_Toc374431930)

[5. Comments 8](#_Toc374431931)

[6. Regions 9](#_Toc374431932)

[7. Blocks 10](#_Toc374431933)

[8. White space 11](#_Toc374431934)

[8.1. Blank lines 11](#_Toc374431935)

[8.2. Other spaces 11](#_Toc374431936)

[9. Naming conventions 12](#_Toc374431937)

[9.1. Class naming 12](#_Toc374431938)

[9.2. Interface naming 12](#_Toc374431939)

[9.3. Generic classes naming 12](#_Toc374431940)

[9.4. Enum naming 12](#_Toc374431941)

[9.5. Read-only and const naming 13](#_Toc374431942)

[9.6. Parameter naming 13](#_Toc374431943)

[9.7. Local variable naming 13](#_Toc374431944)

[9.8. Instance private variables 13](#_Toc374431945)

[9.9. Method names 14](#_Toc374431946)

[9.10. Property names 14](#_Toc374431947)

[9.11. Events 14](#_Toc374431948)

[9.12. Exceptions 15](#_Toc374431949)

[10. Code structure and logic 16](#_Toc374431950)

# About this document

This document describes various standards that should be followed while developing standalone applications within NetRom organization.

# Files

## Project files, Solution files and namespaces

Use separate folders for:

* Solution file
* Projects within the solution (name the projects as CompanyName.ProjectName). The project name must match with the assembly name and with the namespace.
* Modules within each project (name the modules as CompanyName.ProjectName.ModuleName)
* Logically grouped files

Do not place projects folders within other projects, always place projects in the root of the solution folder.

Always match the namespace with the project name.

Use Pascal casing for Solution, Project and Module folders and namespaces. Separate with . (dot) the logical levels (company from project, from module).

Example:

|  |  |  |
| --- | --- | --- |
| **Folder** | **Name** | **Namespace** |
| Solution | CompanyName.ProjectName | - |
| - Project1 | CompanyName.Project1 | CompanyName.Project1 |
| - Module1 | CompanyName.Project1.Module1 | CompanyName.Project1.Module1 |
| - Module2 | CompanyName.Project1.Module2 | CompanyName.Project1.Module2 |
| - Generics | CompanyName.Project1.Generics | CompanyName.Project1.Generics |
| - Project2 | CompanyName.Project2 | CompanyName.Project2 |
| - Module1 | CompanyName.Project2.Module1 | CompanyName.Project2.Module2 |
| - CompanyName.Project.sln | - | - |

## Class files

Place one class/interface definition per file. The filename should be the same as the class name. Sometimes additional small class definitions per file are allowed if they are very tightly connected to the main class and only used by the main class or together with the main class.

Place the definition of a nested class in a separate file and make the parent class partial. Name the file ParentClass.NestedClass.<extension>.

All classes must be enclosed within a namespace (explicitly specified in each file).

Namespace should contain at least the company and project names. In case the project is organized in modules, the module should also be reflected in the namespace.

# Indentation

## Wrap long lines

Break an expression when it doesn’t fit on a single line and the line is too big.

Break:

* After a comma
* After an operator

Apply correct indentation on the next line.

Declare variables one per line.

Example:

int result = a + b + c \* (a - b \* c) + (2 \* a - 2 \* b) \*

(a - b);

string.Format("{0} - {1} - {2} - {3}",

a,

b,

Math.Sin(a) + Math.Sin(b),

Math.Cos(a) + Math.Cos(b));

## Indentation characters

Use only tabs for indentation instead of spaces. Use only one tab per indent level. Use a tab size of 4.

# Comments

For the .NET languages use the XML documentation system introduced by Microsoft whenever possible (e.g. for class descriptions) (in C# lines must start with “///”).

For class and method description, use the XML documentation:

/// <summary>

/// This is a sample class

/// </summary>

public class SampleClass

{

/// <summary>

/// Constructor

/// </summary>

public SampleClass()

{

}

/// <summary>

/// This method computes the sum of two integers

/// </summary>

/// <param name="a">The first operand</param>

/// <param name="b">The second operand</param>

/// <returns>The sum of the two operands</returns>

public int Sum(int a, int b)

{

return a + b;

}

}

Comment statements and variables using single line comments as much as possible aligned (multiple lines can be commented at once in IDEs using simple key combinations):

int[] numbers = new int[10]; //an array of numbers

for (int i = 0; i < numbers.Length; i++) //iterate the array of numbers

numbers[i] = i; //set the integer at position i

# Regions

Use code regions to group methods, properties, types, variable declarations and also very large logical sections of code (e.g. algorithms). Nested regions may also be used to create subgroups.

Group the functions based on their scope (private, protected…) and create regions.

Create separate regions for interface implementations, event handlers, overridden members, abstract members, static members.

Example:

#region private variables

private int \_x;

private int \_y;

#endregion

#region properties

public int X

{

get { return \_x; }

set { \_x = value; }

}

public int Y

{

get { return \_y; }

set { \_y = value; }

}

#endregion

# Blocks

Place the block start string (the opening brace in C# - “{“) at the beginning of a new line after a declaration or statement and without indent.

Place the block end string (the closing brace in C# - “}”) at the end of the block on a new line.

Example:

class Block

{

int x;

Block()

{

x = 0;

}

}

Generally it is a good idea to use blocks even if there is a single statement in IFs, FORs, WHILEs and DO-WHILEs and other flow control statements. This avoids logical errors when there is a need for additional inner statements and a block is not created.

# White space

## Blank lines

Blank lines should be used between:

* Logical sections of code
* Class and interface definitions
* Methods
* Properties

Blank lines improve readability in general. But a class with collapsed members seems sparse when members are separated by empty lines.

## Other spaces

Place a single space after a comma or a semicolon:

Sum(a, b);

Surround operators with single spaces:

return a + b;

# Naming conventions

Use good names for the variables, methods and parameters that describe the semantics. Do not use very long names like: AddUserToCustomCollectionAndRefresh(userToBeAddedToCustomCollection)

Do not use Hungarian notation of variables.

## Class naming

Use Pascal casing (capitalize the first letter of each word). Names must be nouns or noun phrases.

Good:

class SomeClass

Bad:

class some\_class

class someClass

class someclass

class SOMECLASS

## Interface naming

Use Pascal casing and prefix the name with “I”. Names must be nouns or adjectives describing behavior.

Good:

interface IMovingObject

Bad:

interface iMovingObject

interface Imovingobject

interface MovingObjectInterface

## Generic classes naming

Use T as TypeName identifier:

public class List<T> : IList<T>

{

}

## Enum naming

Use Pascal casing for enum type names and enum value names. Do not prefix or suffix type names or value names.

Good:

enum Direction

{

Horizontal,

Vertical

}

Bad:

enum Direction

{

HORIZONTAL,

VERTICAL

}

enum direction

{

horizontal,

vertical

}

## Read-only and const naming

Use upper case names, separate words with “\_”.

Good:

const int DEFAULT\_SIZE = 4;

Bad:

const int DEFAULTSIZE = 4;

const int DefaultSize = 4;

## Parameter naming

Use Camel casing (capitalize the first letter of each word except the first one).

## Local variable naming

Use Camel casing.

Indexers variables generally should be named “i”, ”j”, “k” in trivial counting loops. But in case of more complex loops give them meaningful names.

Avoid single-character and enumerated names for variables.

Good:

int itemIndex;

Bad:

int ItemIndex;

## Instance private variables

Use Camel casing and precede the name with an “\_”.

Good:

private int \_x;

Bad:

private int x;

## Method names

Use Pascal casing.

Good:

public object CreateObject();

Bad:

public object createObject();

public object create\_object();

## Property names

Use Pascal naming. If a property has a backing field, name the property like the backing field.

Good:

private int \_lineNumber;

public int LineNumber

{

get { return \_lineNumber; }

set { \_lineNumber = value; }

}

Bad:

private int \_lineNumber;

public int lineNumber

{

get { return \_lineNumber; }

set { \_lineNumber = value; }

}

Never prefix with Get and Set.

The name of the property/backing field should have a meaning and when possible to suggest the type:

Good:

private bool \_isReadOnly;

public bool IsReadOnly

{

get { return \_isReadOnly; }

set { \_isReadOnly = value; }

}

Bad:

private bool \_readOnly;

public bool ReadOnly

{

get { return \_readOnly; }

set { \_readOnly = value; }

}

## Events

Use Pascal casing.

Name event handlers with “EventHandler” suffix. Name event argument classes with “EventArgs” suffix.

Good:

public class FormUpdateEventArgs : EventArgs

public delegate void FormUpdateEventHandler(object sender, FormUpdateEventArgs e)

Bad:

public class FormUpdateArguments : EventArgs

public delegate void FormUpdate (object sender, FormUpdateEventArgs e)

## Exceptions

Use Pascal casing.

Name exception types with “Exception” suffix.

Good:

public class SomeErrorException : Exception

Bad:

public class SomeError : Exception

# Code structure and logic

* Do not use magic numbers or strings in your code. Use constants instead.
* Do not use functions that spread over too many lines. Use many smaller, flexible functions instead.
* Do not use functions with too many parameters (more than 7). When too many, try to group them in structures or classes.
* Keep in mind the reusability when creating your classes and functions. Any piece of code that may be used in more than one place should be created is such a way that can be easy reused.

Try to separate as much as possible the layout related code, by the model of your data.