**1. Introduction**  
This document provides the C# coding standards for development. The goal is to define guidelines to enforce a consistent style and formatting and help developers avoid common pitfalls and mistakes.

Consistency is the key to maintainable code. This statement is most true for naming your projects, source files, and identifiers including variables, properties, methods, parameters, classes, interfaces and namespaces.

|  |  |  |
| --- | --- | --- |
| **Identifier Type** | **Rules for Naming** | **Example** |
| Namespaces | Pascal Case; should start with the company namespace, followed by application "layers" as applicable | CoCon.DataServices.Contacts |
| Classes | Nouns in Pascal Case (also use for structs) | class Contact |
| Exception Classes | Pascal Case and always end with the suffix Exception | Class CoConGeneralException |
| Interfaces | Pascal Case prefixed by "I" | interface ICustomer |
| Methods | Verb or Verb/Noun pair in Pascal Case | GetBackground(); |
| Properties | Pascal Case; do not prefix with get or set | FirstName |
| Constants | All uppercase with words separated by underscores | public const int MIN\_WIDTH = 4; |
| Enum Type | Hungarian notation. | enuHatType |
| Enum Value | All uppercase with words separated by underscores | BOWLER  TOP\_HAT |
| Instance Variable | Camel Case, prefixed with "m"; always make private and accessible via a property with the applicable scope. | private int mCurrentWidth; |
| Local Variable | Camel Case.  Always name the variable appropriately, for example what it is being used for.  Avoid using single characters like "x" or "y" except in FOR loops. **Never name a variable the same as the type.** | // Wrong  AppUser appUser;  // Correct  AppUser appUserData; |
| Parameters | Camel Case  Never name a parameter the same as the type. | public void GetContact(int contactId); |
| Web Page | Pascal Case (Very important since Visual Studio automatically creates Code Behind class on this name.) | ShowResults.aspx |
| Web page controls | Camel Case with control type prefix. For user controls, prefix an appropriately named control with "uc". | lbFirstName  txtFirstName  ddCountry  ucPopupDisplay |

**2.1 Case sensitivity**

To avoid confusion regarding the use of case sensitivity, **do not create** any of the following:

* Two namespaces with names that differ only by case.
* A method with parameter names that differ only by case.
* Methods with names that only differ by a single letter or number.
* A type with property names that differ only by case.
* A type with method names that differ only by case.
* Variables with a type that differs only by case.
* Variables matching a data type using a different case.

**3. Coding Style**  
Consistent layout, format, and organization are keys to creating maintainable code. The following sections describe the preferred way to implement source code to create readable, clear, and consistent code that is easy to understand and maintain.  
  
**3.1 Formatting**

* Never declare more than 1 namespace per file.
* Avoid putting multiple classes in a single file.
* Always place curly braces ({ and }) on a new line.
* Always use curly braces ({ and }) in conditional statements.  
    
  **Example**

if (contactCount == 0)  
{          
  // code for true case       
}  
else  
{  
   // code for false case  
}

* Always put parenthesis around all clauses in if statements.  
    
  **Example**  
    
  **Bad**  
    
  if (contactCount == 0 && retrievingData)  
  {  
  }  
    
  **Good**  
    
  if ((contactCount == 0) &&  
  (retrievingData))  
  {  
  }
* Never make method calls within a clause of a if statement  
    
  **Example**  
    
  **Bad**  
    
  if (GetContacts().Count == 0)  
  {  
  }  
    
  **Good**  
    
  List<Contact> contactList = GetContacts();  
  if (contactList.Count == 0)  
  {  
  }
* Do not use elseif to string if statements together.
* Always use a Tab and Indention size of 2, inserted as spaces.
* Declare each variable independently, not in the same statement.  
    
  **Example**  
    
  **Bad**  
    
  string firstName, lastName;  
    
  **Good**  
    
  string firstName  
  string lastName;
* Place namespace "using" statements together at the top of the file. Group .NET namespaces above custom namespaces. Organize these namespaces alphabetically. NOTE: Visual Studio provides tools to sort using statements in the desired order.
* Group internal class implementation by type in the following order using #region statements:
  + Attributes
  + Constructors
  + Properties
  + Methods
* Order declarations within type groups based upon access modifier and visibility:
  + Public
  + Protected
  + Internal
  + Private

* Order methods within the public/protected/private sections alphabetically.
* Segregate interface implementation using #region statements.
* Recursively indent all code blocks contained within braces. NOTE: Visual Studio supports formatting code by Edit/Advanced/Format Code (or control-K, control-D). This requires setting the editor tabbing and spacing as indicated above.
* Use white space (CR/LF, Tabs, and so on) liberally to separate and organize code.
* Avoid declaring multiple attribute declarations within a single line. Instead stack each attribute as a separate declaration. This is required in all scopes (assembly, type, method, member, parameter, and so on).  
    
  **Example**  
    
  **Bad**  
    
  [Attribute1] [Attribute1] [Attribute1]  
  publicclassMyClass  
  {  
  }  
    
  **Good**  
    
  [Attribute1]  
  [Attribute1]  
  [Attribute1]  
  publicclassMyClass  
  {  
  }
* The "this" and "base" keywords should be used when needed to specify the namespace of an object used, and not as part of general practice.
* If in doubt, always err on the side of clarity and consistency.
* Parameters in method declarations should be on separate lines and all parameters should be left-aligned.  
    
  **Example**  
    
  publicstaticvoid Update(Contact contactData,  
  IDbConnection dbConn,  
  IDbTransaction dbTrans)  
  {  
  }
* Parameters in method calls should be on separate lines with all parameters left-aligned.  
    
  **Example**  
    
  Contact.Update(contactData,  
  dbConn,  
  dbTrans);
* Do not insert spaces in a parameter list for method declarations or method calls NOTE: This behavior is configurable in Visual Studio.  
    
  **Example**  
    
  Contact.Update(contactData,  
  dbConn,  
  dbTrans);  
    
  Rounded Rectangular Callout: Rounded Rectangular   
  Do not insert spaces here Callout: Or here
* Declare all variables at the beginning of a method. Do not place declarations throughout the method body.
* Place a "using" or "imports" statement at the beginning of each class to avoid having to type a fully qualified namespace/class/method name to invoke a method:  
    
  **Example**  
  **Bad**  
    
  CoCon.SystemFramework.LoggerServices.Write(â€¦);  
    
  **Good**  
    
  using CoCon.SystemFramework;  
  â€¦  
  LoggerServices.Write(â€¦);
* Put parenthesis around the entire entity being cast.  
    
  **Example**  
  enuHatType hatType = (enuHatType)(DBService.GetInt(dataRow,  
  COLUMN\_NAME,  
  DEFAULT\_VALUE));
* A parent class is responsible for updating the relationship table with its child/included objects, its child objects do not address that relationship.

**3.2 Commenting**

* All comments should be written in U.S. English.
* Use // or /// but do NOT use /\* \*/.
* Commented out code should be code that will be soon deleted, and marked with a TODO:.
* Do not "flowerbox" comment blocks.  
    
  **Example**  
    
  **Bad**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Comment block

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
**Good**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Comment block

* Use inline-comments to explain assumptions, known issues, and algorithm insights.
* Do not use inline-comments to explain obvious code. Well written code is self documenting.
* Use "TODO:" task list tokens to mark incomplete methods or sections.  
    
  **Example**

// TODO: handle validation errors

* Do NOT use "HACK" or "UNDONE" task list tokens. Other tokens may be added with team consensus and they are added to this document to define how they are to be used.
* Always apply C# comment-blocks (///) to classes. Class comment block should include the following.
  + **Required**
    - Summary (<summary>): used to define what the class is used for
  + **Optional**
    - Remarks (<remarks>): used to provide further explanation of the class use

* Always apply C# documentation comment blocks (///) to constructors, properties, and methods. Comment blocks should include the following sections.
  + **Required**
    - Summary (<summary>): used to define what the constructor, property, or method is used for.
  + **Conditional**
    - Param (<param name="theNameOfTheParameter">): Required for each parameter passed to the constructor or method. Should include what the parameter its primary use.
    - Returns (<returns>): Required for any method that returns a value. Should describe the returned data type and a description of what it is set to.

* + **Optional**
    - Remarks (<remarks>): Used to convey more information about the method. This may include pseudo-code for complicated methods.
    - Exception (<exception cref="CoConGeneralException">): Used to indicate what exception(s) are potentially thrown by the method.
* Additional C# document comment blocks guidelines
  + Add a blank line between the sections for readability.
  + Sections should be ordered as follows:
    - Summary
    - Param
    - Return
    - Remark
    - Exception

**3.3 Visual Studio .Net Environment Setup**

* Select Tools/Options
* Select Text Editor
* Select All Languages/Tabs and set the following:
  + Indenting: Block
  + Tab Size: 2
  + Tab Indent Size: 2
  + Select "Insert Spaces"

* Select C#/Formatting/General: check all checkboxes
* Select C#/Formatting/Indentation: check all checkboxes except "Indent open and close braces" that should be unchecked.
* Select C#/Formatting/New Lines: check all checkboxes

**4. Language Usage**  
  
**4.1 Variables and Types**

* Do not omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.  
    
  **Example**

// Bad!

void WriteEvent(string message)

{

}

// Good!

privatevoid WriteEvent(string message)

{

}

* Declare all variables at the beginning of a method and initialize object variables to null. Scalar variables do not need to be initialized at declaration.
* Use the simplest data type, list, or object required. For example, use int over long unless you know you need to store 64 bit values.
* Always use the built-in data type aliases, not the .NET common type system (CTS).  
    
  **Example**
  + short not System.Int16
  + int not System.Int32
  + long not System.Int64
  + string not System.String
* Declare instance variables as private. Use properties to provide access to them with public, protected, or internal access modifiers as applicable.
* enum: should be int unless you have an explicit need for long.
* Avoid using inline numeric literals (magic numbers). Instead, use a constant or enum.
* Avoid declaring inline string literals. Instead use constants, resources or other data sources.
* Use the @ prefix when initializing string constants or variables instead of escaped strings.  
    
  **Example**  
    
  **Bad**  
  private const string PATH = "C:\\Projects\\CoCon\\";  
    
  **Good**  
    
  private const string PATH = @"C:\Projects\CoCon\";
* For string template construction, use String.Format()
* For string concatenation being performed more than once on a string, use StringBuilder
* Avoid concatenating strings inside a loop. Use StringBuilder if concatenation is required inside the loop.
* Do not compare strings to String.Empty or "" to check for empty strings. Use String.IsNullOrEmpty or myString.Length == 0
* When comparing one string to another, use myString1.Equals(myString2)
* Avoid hidden string allocations, especially within a loop. NOTE: Strings are inmutable so changing the value of a string actually allocates a new string.
* Do not invoke methods in conditional statements
* Avoid creating recursive methods. Any perceived need for a recursive method must be reviewed by the team before implementation.

**4.2 Exception Handling**

* Do not use try/catch blocks for flow-control.
* Only catch exceptions when you can add value such as context information to the exception.
* Never declare an empty catch block. This results in "dumping the exception on the floor".
* Avoid nesting a try/catch within a catch block. Any perceived need for nested try/catch blocks must be reviewed by the team before implementation.
* Use exception filters where possible.  
    
  **Example**

catch(CoConGeneralException exc )

{

  //do actions

}

catch(Exception exc)

{

  //do actions

}

* Order exception filters from most to least derived exception type.
* Avoid re-throwing an exception. Allow it to bubble-up if you can add no additional data to help with the debugging.
* Never use "e" as the variable for the exception. Use "exc" or an abbreviated version of the exception being caught.
* If re-throwing an exception, omit the exception argument from a throw statement to preserve the original call stack.  
    
  **Example**

// Bad

catch(Exception exc)

{

Log(exc);

throw exc;

}

// Good

catch(Exception exc)

{

Log(exc);

throw;

}

* Only use the finally block to release resources from a try statement.
* When retrieving a DataTable, and expecting only zero or one rows, throw an Exception if the Rows.Count value is anything else  
    
  **Example**

dataTable = DBService.ExecuteQuery(sql);

switch (dataTable.Rows.Count)

{

   case 0:

     // process no rows case

    break;

  case 1:

    // process single row case

  break;

 default:

    ArgumentException exc = newArgumentException("Error message here");

  throw exc;  
}

**5. Database**  
Given that data classes will be using inline, dynamic, parameterized SQL for database access, the following sets of guidelines will be followed.

* Use the SELECT constant to select columns. Do not use "SELECT \*".
* Always declare all column names as constants.
* Always use parameters in a SQL statement instead of inline variables for all data.
* When transactions are used, each Begin must be matched with a Rollback/Commit.
* For complex statements, please feel free to consult a DBA for query optimization advice.