C#

**Coding Standards for INSS**

**Document Version 1.0**

**October 2018**

Copyright © INSS 2018

All Rights Reserved

**License**

**Usage & Distribution**

Publication of the work or derivation of the work in any form is prohibited unless a valid license is obtained from the copyright holder.

Commercial redistribution of the work or derivative of the work in any form is prohibited unless a valid license is obtained from the copyright holder.

Non-commercial redistribution of the work is permitted in its original unmodified form so long as all license and all copyright notices are retained and respected.

**Trademarks**

All uses of terms that are known trademarks or service marks have been appropriately capitalized. The Publisher cannot attest to the accuracy of this information. Use of terms within this work should not be regarded as affecting the validity of any trademark or service mark. All Trademarks are the property of their respective owners.

**Disclaimer**

The information provided is on an “As-Is” basis. Use at your own risk. The Author and Publisher shall have neither liability nor responsibility to any person or entity with respect to the loss or damages arising from the information contained in this work. This work may include inaccuracies or typographical errors and solely represent the opinions of the Author. Changes are periodically made to this document without notice.

Any action related to this work will be governed by Texas state law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

The Author reserves the right to revise these terms or terminate the above license at any time without notice.

**1. Introduction**

This document describes rules and recommendations for developing applications and class libraries using the C# Language. The goal is to define guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes.

Specifically, this document covers *Naming Conventions*, *Coding Style*, *Language Usage*, and *Object Model Design*.

**1.1 Scope**

This document only applies to the C# Language and the .NET Framework Common Type System(CTS) it implements. Although the C# language is implemented alongside the .NET Framework, this document does not address usage of

.NET Framework class libraries. However, common patterns and problems related to C#’s usage of the .NET Framework are addressed in a limited fashion.

Even though standards for curly-braces ({ or }) and white space(tabs vs. spaces) are always controversial, these topics are addressed here to ensure greater consistency and maintainability of source code.

**1.2 Document Conventions**

Much like the ensuing coding standards, this document requires standards in order to ensure clarity when stating the rules and guidelines. Certain conventions are used throughout this document to add emphasis.

Below are some of the common conventions used throughout this document.

|  |  |
| --- | --- |
| Coloring & Emphasis: |  |
| Blue | Text colored blue indicates a C# keyword or .NET type. |
| **Bold** | Text with additional emphasis to make it stand-out. |
| Keywords: |  |
| **Always** | Emphasizes this rule must be enforced. |
| **Never** | Emphasizes this action must not happen. |
| **Do Not** | Emphasizes this action must not happen. |
| **Avoid** | Emphasizes that the action should be prevented, but |
|  | some exceptions may exist. |
| **Try** | Emphasizes that the rule should be attempted whenever possible and appropriate. |
| **Example** | Precedes text used to illustrate a rule or recommendation. |
| **Reason** | Explains the thoughts and purpose behind a rule or recommendation. |

**1.3 Terminology & Definitions**

The following terminology is referenced throughout this document:

**Access Modifier**

C# keywords public, protected, internal, and private declare the allowed code-accessibility of types and their members. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public.

**Camel Case**

A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized. **Example**: customerName

**Common Type System**

The .NET Framework common type system (CTS) defines how types are declared, used, and managed. All native C# types are based upon the CTS to ensure support for cross-language integration.

**Identifier**

A developer defined token used to uniquely name a declared object or object instance.

**Example**: public class **MyClassNameIdentifier**{ … }

**Magic Number**

Any numeric literal used within an expression (or to initialize a variable) that does not have an obvious or well- known meaning. This usually excludes the integers 0 or 1 and any other numeric equivalent precision that evaluates as zero.

**Pascal Case**

A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized. **Example**: CustomerName

**Premature Generalization**

As it applies to object model design; this is the act of creating abstractions within an object model not based upon concrete requirements or a known future need for the abstraction. In simplest terms: “Abstraction for the sake of Abstraction.”

**1.4 Flags**

The following flags are used to help clarify or categorize certain statements:

***[C#v2+]***

A flag to identify rules and statements that apply only to C# Language Specification v2.0 or greater.

Quick Summary

This section contains tables describing a high-level summary of the major standards covered in this document. These tables are not comprehensive, but give a quick glance at commonly referenced elements.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | | |  | | | | | | | | | |
|  | **“c”** | | | = |  | | | |  | | | |  | |  | | | |
|  | **“P”** | | | = |  | | | |  | | | |  | |  | | | |
|  | **“\_”** | | | = |  | | | | |  | |  | | | | | |
|  | **“**x**”** | | | = |  | | | |  | | | |  | |  | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | **Identifier** | | |  |  | **Public** | | **Protected** | | | **Internal** | | | **Private** | | **Notes** | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Project File | | | |  | **P** | | x | | | x | | | x | | Match Assembly & Namespace. | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Source File | | | |  | **P** | | x | | | x | | | x | | Match contained class. | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Other Files | | | |  | **P** | | x | | | x | | | x | | Apply where possible. | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Namespace | | | |  | **P** | | x | | | x | | | x | | Partial Project/Assembly match. | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Class or Struct | | | |  | **P** | | **P** | | | **P** | | | **P** | | Add suffix of subclass. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Interface | | |  |  | **P** | | **P** | | | **P** | | | **P** | | Prefix with a capital I. | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Generic Class | | | |  | **P** | | **P** | | | **P** | | | **P** | | Use T or K as Type identifier. | | | |
|  | ***[C#v2+]*** | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Method | | |  |  | **P** | | **P** | | | **P** | | | **P** | | Use a Verb or Verb-Object pair. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Property | | |  |  | **P** | | **P** | | | **P** | | | **P** | | Do not prefix with Get or Set. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Field | | |  |  | **P** | | **P** | | | **P** | | | **\_c** | | Only use Private fields. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | | **No Hungarian Notation!** | | | |
|  | Constant | | |  |  | **P** | | **P** | | | **P** | | | **\_c** | |  | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Static Field | | | |  | **P** | | **P** | | | **P** | | | **\_c** | | Only use Private fields. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Enum | | |  |  | **P** | | **P** | | | **P** | | | **P** | | Options are also PascalCase. | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Delegate | | |  |  | **P** | | **P** | | | **P** | | | **P** | |  | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |
|  | Event | | |  |  | **P** | | **P** | | | **P** | | | **P** | |  | | | |
|  |  | | | |  |  | |  | | |  | | |  | |  | | | |
|  | Inline Variable | | | |  | x | | x | | | x | | | **c** | | Avoid single-character and enumerated | | | |
|  |  | | |  |  |  | |  | | |  | | |  | | names. | | | |
|  | Parameter | | |  |  | x | | x | | | x | | | **c** | |  | | | |
|  |  | | |  |  |  | |  | | |  | | |  | |  | | | |

**1.4.2 Coding Style**

|  |  |
| --- | --- |
| **Code** | **Style** |
| Source Files | One Namespace per file and one class per file. |
| Curly Braces | On new line. Always use braces when optional. |
| Indention | Use tabs with size of 4. |
| Comments | Use // or /// |
| Variables | One variable per declaration. |

**1.4.3 Language Usage**

|  |  |
| --- | --- |
| **Code** | **Style** |
| **Native Data Types** | Use built-in C# native data types vs .NET CTS types. |
|  | (Use int NOT Int32) |
| **Enums** | Avoid changing default type. |
| **Generics *[C#v2+]*** | Prefer Generic Types over standard or strong-typed classes. |
| **Properties** | Never prefix with Get or Set. |
| **Methods** | Use a maximum of 7 parameters. |
| **base**and **this** | Use only in constructors or within an override. |
| **Ternary conditions** | Avoid complex conditions. |
| **foreach statements** | Do not modify enumerated items within a foreach statement. |
| **Conditionals** | Avoid evaluating Boolean conditions against true or false. |
|  | No embedded assignment. |
|  | Avoid embedded method invocation. |
| **Exceptions** | Do not use exceptions for flow control. |
|  | Use throw; not throw e; when re-throwing. |
|  | Only catch what you can handle. |
|  | Use validation to avoid exceptions. |
|  | Derive from Execption not ApplicationException. |
| **Events** | Always check for null before invoking. |
| **Locking** | Use lock() not Monitor.Enter(). |
|  | Do not lock on an object type or “this”. |
|  | Do lock on private objects. |
| **Dispose()**& **Close()** | Always invoke them if offered, declare where needed. |
| **Finalizers** | Avoid. |
|  | Use the C# Destructors. |
|  | Do not create Finalize() method. |
| **AssemblyVersion** | Increment manually. |
| **ComVisibleAttribute** | Set to false for all assemblies. |

**2. Naming Conventions**

Consistency is the key to maintainable code. This statement is most true for naming your projects, source files, and identifiers including Fields, Variables, Properties, Methods, Parameters, Classes, Interfaces, and Namespaces.

**2.1 General Guidelines**

1.Always use Camel Case or Pascal Case names.

2.Avoid ALL CAPS and all lowercase names. Single lowercase words or letters are acceptable.

3.Do not create declarations of the same type (namespace, class, method, property, field, or parameter) and access modifier (protected, public, private, internal) that vary only by capitalization.

4.Do not use names that begin with a numeric character.

5.Do add numeric suffixes to identifier names.

6.Always choose meaningful and specific names.

7.Always err on the side of verbosity not terseness.

8.Variables and Properties should describe an entity not the type or size.

9.Do not use Hungarian Notation! **Example**: strName or iCount

10.Avoid using abbreviations unless the full name is excessive.

11.Avoid abbreviations longer than 5 characters.

12.Any Abbreviations must be widely known and accepted.

13.Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.

14.Do not use C# reserved words as names.

15.Avoid naming conflicts with existing .NET Framework namespaces, or types.

16.Avoid adding redundant or meaningless prefixes and suffixes to identifiers

**Example:**

// Bad!

public enum ColorsEnum {…}

public class CVehicle {…}

public struct Rectangle*Struct* {…}

17.Do not include the parent class name within a property name. **Example:**Customer.Name NOT Customer.CustomerName

18.Try to prefix Boolean variables and properties with “Can”, “ Is” or “ Has”.

19.Append computational qualifiers to variable names like Average, Count, Sum, Min, and Max where appropriate.

20.When defining a root namespace, use a Product, Company, or Developer Name as the root. **Example:**

SISSMO.StringUtilities

**2.2 Name Usage & Syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier** | **Naming Convention** | |  |
|  |  | |  |
| **Project File** | Pascal Case. | |  |
|  | Always match Assembly Name & Root Namespace. | | |
|  | **Example:** | |  |
|  | SISSMO.Web.csproj -> SISSMO.Web.dll -> namespace | | |
|  | SISSMO.Web | |  |
|  |  | |  |
| **Source File** | Pascal Case. | |  |
|  | Always match Class name and file name. | | |
|  | Avoid including more than one Class, Enum (global), or Delegate (global) per file. Use a | | |
|  | descriptive file name when containing multiple Class, Enum, or Delegates. | | |
|  | **Example:** | |  |
|  | MyClass.cs => | | public class MyClass |
|  |  | | {…} |
|  |  | |  |
| **Resource** | Try to use Pascal Case. | |  |
| or |  | |  |
| **Embedded File** | Use a name describing the file contents. | | |
|  |  | |  |
| **Namespace** | Pascal Case. | |  |
|  | Try to partially match Project/Assembly Name. | | |
|  | **Example:** | |  |
|  | namespace SISSMO.Web | | |
|  | {…} | |  |
|  |  | |  |
| **Class or Struct** | Pascal Case. | |  |
|  | Use a noun or noun phrase for class name. | | |
|  | Add an appropriate class-suffix when sub-classing another type when possible. | | |
|  | **Examples:** | |  |
|  | private class MyClass | | |
|  | {…} | |  |
|  | internal class SpecializedAttribute : Attribute | | |
|  | {…} | |  |
|  | public class CustomerCollection : CollectionBase | | |
|  | {…} | |  |
|  | public class CustomEventArgs : EventArgs | | |
|  | {…} | |  |
|  | private struct ApplicationSettings | | |
|  | {…} | |  |
|  |  | |  |
| **Interface** | Pascal Case. | |  |
|  | Always prefix interface name with capital “I”. | | |
|  | **Example:** | |  |
|  | interface ICustomer | | |
|  | {…} | |  |
|  |  | |  |
|  | | |  | |

|  |  |  |
| --- | --- | --- |
|  | **Generic Class** | Always use a single capital letter, such as T or K. |
|  | & | **Example:** |
|  |  | public class FifoStack<T> |
|  | **Generic** | { |
|  | **Parameter Type** | public void Push(<T> obj) |
|  | {…} |
|  |  |
|  | ***[C#v2+]*** | public <T> Pop() |
|  |  | {…} |
|  |  | } |
|  |  |  |
|  | **Method** | Pascal Case. |
|  |  | Try to use a **Verb**or **Verb-Object**pair. |
|  |  | **Example:** |
|  |  | public void Execute() {…} |
|  |  | private string GetAssemblyVersion(Assembly target) {…} |
|  |  |  |
|  | **Property** | Pascal Case. |
|  |  | Property name should represent the entity it returns. Never prefix property names with |
|  |  | “Get” or “ Set”. |
|  |  | **Example:** |
|  |  | public string Name |
|  |  | { |
|  |  | get{…} |
|  |  | set{…} |
|  |  | } |
|  |  |  |
|  | **Field** | Pascal Case. |
|  |  | Avoid using non-private Fields! |
|  | (Public, Protected, | Use Properties instead. |
|  | or Internal) | **Example:** |
|  |  |
|  |  | public string Name; |
|  |  | protected IList InnerList; |
|  |  |  |
|  | **Field**(Private) | Camel Case and prefix with a single underscore (\_) character. |
|  |  | **Example:** |
|  |  | private string \_name; |
|  |  |  |
|  | **Constant**or | Treat like a Field. |
|  | **Static Field** | Choose appropriate Field access-modifier above. |
|  |  |  |
|  | **Enum** | Pascal Case (both the Type and the Options). |
|  |  | Add the FlagsAttribute to bit-mask multiple options. |
|  |  | **Example:** |
|  |  | public enum CustomerTypes |
|  |  | { |
|  |  | Consumer, |
|  |  | Commercial |
|  |  | } |
|  |  |  |

|  |  |
| --- | --- |
| **Delegate**or **Event** | Treat as a Field. |
|  | Choose appropriate Field access-modifier above. |
|  | **Example:** |
|  | public event EventHandler LoadPlugin; |
|  |  |
| **Variable**(inline) | Camel Case. |
|  | Avoid using single characters like “x” or “ y” except in FOR loops. |
|  | Avoid enumerating variable names like text1, text2, text3 etc. |
|  |  |
| **Parameter** | Camel Case. |
|  | **Example:** |
|  | public void Execute(string commandText, int iterations) |
|  | {…} |
|  |  | |

**3. Coding Style**

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are key to creating maintainable code. The following sections describe the preferred way to implement C# source code in order to create readable, clear, and consistent code that is easy to understand and maintain.

**3.1 Formatting**

1.Never declare more than 1 namespace per file.

2.Avoid putting multiple classes in a single file.

3.Always place curly braces ({ and }) on a new line.

4.Always use curly braces ({ and }) in conditional statements.

5.Always use a Tab & Indention size of 4.

6.Declare each variable independently – not in the same statement.

7.Place namespace “using” statements together at the top of file. Group .NET namespaces above custom namespaces.

8.Group internal class implementation by type in the following order:

a.Member variables.

b.Constructors & Finalizers.

c.Nested Enums, Structs, and Classes.

d.Properties

e.Methods

9.Sequence declarations within type groups based upon access modifier and visibility:

a.Public

b.Protected

c.Internal

d.Private

10.Segregate interface Implementation by using #region statements.

11.Append folder-name to namespace for source files within sub-folders.

12.Recursively indent all code blocks contained within braces.

13.Use white space (CR/LF, Tabs, etc) liberally to separate and organize code.

14.Only declare related attribute declarations on a single line, otherwise stack each attribute as a separate declaration.

**Example:**

// Bad!

[Attrbute1, Attrbute2, Attrbute3]

public class MyClass

{…}

// Good!

[Attrbute1, RelatedAttribute2]

[Attrbute3]

[Attrbute4]

public class MyClass {…}

15.Place Assembly scope attribute declarations on a separate line.

16.Place Type scope attribute declarations on a separate line.

17.Place Method scope attribute declarations on a separate line.

18.Place Member scope attribute declarations on a separate line.

19.Place Parameter attribute declarations inline with the parameter.

20.If in doubt, always err on the side of clarity and consistency.

**3.2 Code Commenting**

21.All comments should be grammatically correct, and contain appropriate punctuation.

22.Use inline-comments to explain assumptions, known issues, and algorithm insights.

23.Do not use inline-comments to explain obvious code. Well written code is self- documenting.

24.Always apply C# comment-blocks (///) to public, protected, and internal declarations.

25.Include comments using Task-List keyword flags to allow comment-filtering.

**Example**:

//TODO: Place Database Code Here

//UNDONE: Removed P\Invoke Call due to errors

//HACK: Temporary fix until able to refactor

26.Only use C# comment-blocks for documenting the API.

27.Always include <summary> comments. Include <param>, <return>, and <exception> comment sections where applicable.

28.Include <see cref=””/> and <seeAlso cref=””/> where possible.

29.For new created code file, make sure to add header comment if the file is not auto-generated.

**Example**:

//////-------------------------------------------------------------------------------------------------------

////// Copyright © INSS 2012-2018

//////

////// Name: CalendarColumn.cs

////// Programmer: Robin Ren

////// Created: 06/14/2017

////// Purpose: Displays date time values in a date time picker for the DataGridView control

////// Modifications: \*RR 06/14/17 SIS-001 (ported from component in custom source code)

////// \*NRA 03/06/18 SIS-002 Add logic for warranties.

//////-------------------------------------------------------------------------------------------------------

30.Make sure to include following information in the comment: name initials of the developer who made this change, modification type/quantity, date, JIRA ticket number, ticket description or other necessary description. The basic pattern is:

//\*name\_initials+line\_qty MM/dd/yy JIRA\_num JIRA\_desc

**Examples:**

1. Developer Robin Ren added 2 code lines:

|  |
| --- |
| /// <summary>  /// Converts a measurement from millimeters to inches  /// </summary>  /// <param name="inches">The value in inches</param>  /// <returns>The value in millimeters, rounded to 8 digits.</returns>  public static float ConvertInchesToMillimeters(float inches)  {  //\*RR+2 06/25/17 SIS-111 Change UOM conversion functions from Decimal to double  double mm = inches \* 25.4;  return (float)System.Math.Round(mm, 8);  } |

1. Add new class, Enum, etc.:

|  |
| --- |
| //\*RTD++ 05/24/17 SVCS-4688 new basecamp metric to keep page count  /// <summary>  /// retrieves the page count  /// </summary>  /// <returns>page count</returns>  private int GetPageCountV2()  {  return ((Document)this.DocumentObject).Pages.Count;  } |

1. Add if/else block:

|  |
| --- |
| string sRetVal = sValue;  //\*DHB+if 12/10/07 WPL003857 Do not attempt if bUOMConversion is false. Just return value as was passed in.  if (bConvertValues && bUOMConversion)  {  // if it's a number, convert to float and then do the conversion  sRetVal = DoFrontEndConversion(bConvertValues, eBackendType, eDisplayType, bUOMConversion, fTol, fValue);  } |

1. Delete code lines:

|  |
| --- |
| //\*BAH+comment 08/16/10 WPL12610 No longer used.  //private void SetSecurityToControl()  //{  //this.purchaseOrderControl1.SetPrintSecurity = this.m\_bPrint;  //this.purchaseOrderControl1.SetElectronicFileSecurity = this.m\_bElectronicFile;  //} |

1. Add lengthy code lines:

|  |
| --- |
| sAnswer = convertVal.ToString("r"); //\*CAA 03/08/12 WPL18881  //\*NRA+start 11/20/07 WPL004004 Fix conversion issues for windowset tab.  //The two dot and two minus patterns should not require numbers  //so they have been changed to look for any characters.  //Regex objTwoDotPattern = new Regex("[0-9]\*[.][0-9]\*[.][0-9]\*");  //Regex objTwoMinusPattern = new Regex("[0-9]\*[-][0-9]\*[-][0-9]\*");  // Regex objTwoDotPattern = new Regex(@".\*\..\*\."); // BDS 04/01/11 WPL015068 Reference member Regex  // Regex objTwoMinusPattern = new Regex(".\*-.\*-"); // BDS 04/01/11 WPL015068 Reference member Regex  //These patterns were allowing a single minus or dot, which could cause issues.  //string strValidRealPattern = "^([-]|[.]|[-.]|[0-9])[0-9]\*[.]\*[0-9]+$";  //string strValidIntegerPattern = "^([-]|[0-9])[0-9]\*$";  //Regex objNumberPattern = new Regex("(" + strValidRealPattern + ")|(" + strValidIntegerPattern + ")");  //This new pattern should match any valid floating point number.  // Regex objNumberPattern = new Regex(@"[-]?[0-9]\*\.?[0-9]+$"); // BDS 04/01/11 WPL015068 Reference member Regex  //\*NRA+end 11/20/07 WPL004004 Fix conversion issues for windowset tab.  break; |

1. Update the existing code line:

|  |
| --- |
| // We found the whole number part of the passed in value  if (wholeval != 0 || decimalstr.Length == 0) //\*CAA 06/18/13 WPL27787 Add or condition to account for 0.0 so this will return 0 in that case  {  retval = wholestr;  } |

**4. Language Usage**

**4.1 General**

1.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!

private Void WriteEvent(string message) {…}

2.Do not use the default (“1.0.\*”) versioning scheme . Increment the AssemblyVersionAttribute value manually.

3.Set the ComVisibleAttribute to false for all assemblies.

4.Only selectively enable the ComVisibleAttribute for individual classes when needed. **Example**:

[assembly: ComVisible(false)]

[ComVisible(true)] public MyClass {…}

5.Consider factoring classes containing unsafe code blocks into a separate assembly.

6.Avoid mutual references between assemblies.

**4.2 Variables & Types**

7.Try to initialize variables where you declare them.

8.Always choose the simplest data type, list, or object required.

9.Always use the built-in C# 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

10.Only declare member variables as private. Use properties to provide access to them with public, protected, or internal access modifiers.

11.Try to use int for any non-fractional numeric values that will fit the int datatype - even variables for non- negative numbers.

12.Only use long for variables potentially containing values too large for an int.

13.Try to use double for fractional numbers to ensure decimal precision in calculations.

14.Only use float for fractional numbers that will not fit double or decimal.

15.Avoid using float unless you fully understand the implications upon any calculations.

16.Try to use decimal when fractional numbers must be rounded to a fixed precision for calculations. Typically this will involve money.

17.Avoid using sbyte, short, uint, and ulong unless it is for interop (P/Invoke) with native libraries.

18.Avoid specifying the type for an enum - use the default of int unless you have an explicit need for long (very uncommon).

19.Avoid using inline numeric literals (magic numbers). Instead, use a Constant or Enum.

20.Avoid declaring string literals inline. Instead use Resources, Constants, Configuration Files, Registry or other data sources.

21.Declare readonly or static readonly variables instead of constants for complex types.

22.Only declare constants for simple types.

23.Avoid direct casts. Instead, use the “as” operator and check for null.

**Example**:

object dataObject = LoadData();

DataSet ds = dataObject as DataSet;

if(ds != null) {…}

24.Always prefer C# Generic collection types over standard or strong-typed collections. ***[C#v2+]***

25.Always explicitly initialize arrays of reference types using a for loop.

26.Avoid boxing and unboxing value types.

**Example**:

|  |  |
| --- | --- |
| int count = 1; |  |
| object refCount = count; | // Implicitly boxed. |
| int newCount = (int)refCount; | // Explicitly unboxed |

27.Floating point values should include at least one digit before the decimal place and one after. **Example**: totalPercent = 0.05;

28.Try to use the “@” prefix for string literals instead of escaped strings.

29.Prefer String.Format() or StringBuilder over string concatenation.

30.Never concatenate strings inside a loop.

31.Do not compare strings to String.Empty or “” to check for empty strings. Instead, compare by using

String.Length == 0.

32.Avoid hidden string allocations within a loop. Use String.Compare() for case-sensitive

**Example**: (*ToLower() creates a temp string*)

// Bad!

int id = -1;

string name = “lance hunt”;

for(int i=0; i < customerList.Count; i++)

{

if(customerList[i].Name.**ToLower()**== name)

{

id = customerList[i].ID;

}

}

// Good!

int id = -1;

string name = “lance hunt”;

for(int i=0; i < customerList.Count; i++)

{

//The “ignoreCase = true” argument performs a

//case-insensitive compare without new allocation.

if(**String.Compare**(customerList[i].Name, name, **true**)== 0)

{

id = customerList[i].ID;

}

}

**4.3 Flow Control**

33.Avoid invoking methods within a conditional expression.

34.Avoid creating recursive methods. Use loops or nested loops instead.

35.Avoid using foreach to iterate over immutable value-type collections. E.g. String arrays.

36.Do not modify enumerated items within a foreach statement.

37.Use the **ternary**conditional operator only for trivial conditions. Avoid complex or compound ternary operations. **Example:**int result = isValid **?**9 **:**4;

38.Avoid evaluating Boolean conditions against true or false.

**Example:**

// Bad!

if (isValid == true) {…}

// Good!

if (isValid) {…}

39.Avoid assignment within conditional statements. **Example:**if((**i=2**)==2) {…}  
40.Avoid compound conditional expressions – use Boolean variables to split parts into multiple manageable expressions.

**Example:**

// Bad!

if (((value > \_highScore) && (value != \_highScore)) && (value < \_maxScore)) {…}

// Good!

isHighScore = (value >= \_highScore); isTiedHigh = (value == \_highScore); isValid = (value < \_maxValue);

if ((isHighScore && ! isTiedHigh) && isValid) {…}

41.Avoid explicit Boolean tests in conditionals.

**Example:**

// Bad!

if(IsValid == true) {…};

// Good!

if(IsValid) {…}

42.Only use switch/case statements for simple operations with parallel conditional logic.

43.Prefer nested if/else over switch/case for short conditional sequences and complex conditions.

44.Prefer polymorphism over switch/case to encapsulate and delegate complex operations.

**4.4 Exceptions**

45.Do not use try/catch blocks for flow-control.

46.Only catch exceptions that you can handle.

47.Never declare an empty catch block.

48.Avoid nesting a try/catch within a catch block.

49.Always catch the most derived exception via exception filters.

50.Order exception filters from most to least derived exception type.

51.Avoid re-throwing an exception. Allow it to bubble-up instead.

52.If re-throwing an exception, preserve the original call stack by omitting the exception argument from the throw statement.

**Example**:

// Bad!

catch(Exception ex)

{

Log(ex);

throw ex;

}

// Good!

catch(Exception ex)

{

Log(ex);

throw;

}

53.Only use the finally block to release resources from a try statement.  
54.Always use validation to avoid exceptions.

**Example**:

//Bad!

try

{

conn.Close();

}

Catch(Exception ex)

{

//handle exception if already closed!

}

//Good!

if(conn.State != ConnectionState.Closed)

{

conn.Close();

}

55.Always set the innerException property on thrown exceptions so the exception chain & call stack are maintained.

56.Avoid defining custom exception classes. Use existing exception classes instead.

57.When a custom exception is required;

a. Always derive from Exception **not**ApplicationException.

b. Always suffix exception class names with the word “Exception”.

c. Always add the SerializableAttribute to exception classes.

d. Always implement the standard “Exception Constructor Pattern”:

public MyCustomException ();

public MyCustomException (string message);

public MyCustomException (string message, Exception innerException);

e. Always implement the deserialization constructor:

protected MyCustomException(SerializationInfo info, StreamingContext contxt);

58.Always set the appropriate HResult value on custom exception classes.

(**Note:**the ApplicationException HResult = -2146232832)

59.When defining custom exception classes that contain additional properties:

a.Always override the Message property, ToString() method and the implicit operator stringto include custom property values.

b.Always modify the deserialization constructor to retrieve custom property values.

c.Always override the GetObjectData(…) method to add custom properties to the serialization collection.

**Example**:

public override void GetObjectData(SerializationInfo info,

StreamingContext context)

{

base.GetObjectData (info, context);

info.AddValue("MyValue", \_myValue);

}

**4.5 Events, Delegates, & Threading**

60.Always check Event & Delegate instances for null before invoking.

61.Use the default EventHandler and EventArgs for most simple events.

62.Always derive a custom EventArgs class to provide additional data.

63.Use the existing CancelEventArgs class to allow the event subscriber to control events.

64.Always use the “lock” keyword instead of the Monitor type.

65.Only lock on a private or private static object.

**Example:**lock(myVariable);  
66.Avoid locking on a Type.

**Example:**lock(typeof(MyClass));

67.Avoid locking on the current object instance.

**Example:**lock(this);

**4.6 Object Composition**

68.Always declare types explicitly within a namespace. Do not use the default “{global}” namespace.

69.Avoid overuse of the public access modifier. Typically fewer than 10% of your types and members will be part of a public API, unless you are writing a class library.

70.Consider using internal or private access modifiers for types and members unless you intend to support them as part of a public API.

71.Never use the protected access modifier within sealed classes unless overriding a protected member of an inherited type.

72.Avoid declaring methods with more than 5 parameters. Consider refactoring this code.

73.Try to replace large parameter-sets (> than 5 parameters) with one or more class or struct parameters – especially when used in multiple method signatures.

74.Do not use the “new” keyword on method and property declarations to hide members of a derived type.

75.Only use the “base” keyword when invoking a base class constructor or base implementation within an override.

76.Consider using method overloading instead of the params attribute (but be careful not to break CLS Compliance of your API’s).

77.Always validate an enumeration variable or parameter value before consuming it. They may contain any value

that the underlying Enum type (default int) supports.

**Example**:

public void Test(BookCategory cat)

{

if (Enum.IsDefined(typeof(BookCategory), cat)) {…}

}

78.Consider overriding Equals() on a struct.

79.Always override the Equality Operator (==) when overriding the Equals() method.

80.Always override the String Implicit Operator when overriding the ToString() method.

81.Always call Close() or Dispose() on classes that offer it.

82.Wrap instantiation of IDisposable objects with a “using” statement to ensure that Dispose() is automatically called.

**Example**:

using(SqlConnection cn = new SqlConnection(\_connectionString)) {…}

83.Always implement the IDisposable interface & pattern on classes referencing external resources.

**Example**: *(shown with optional Finalizer)*

public void Dispose()

{

Dispose(true); GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (disposing)

{

// Free other state (managed objects).

}

//Free your own state (unmanaged objects).

//Set large fields to null.

}

//C# finalizer. (optional)~Base()

{

//Simply call Dispose(false).

Dispose (false);

}

84.Avoid implementing a Finalizer.

Never define a Finalize() method as a finalizer. Instead use the C# destructor syntax.

**Example**

//Good

~MyClass {…}

//Bad

void Finalize(){…}

**5.Object Model & API Design**

1.Always prefer aggregation over inheritance.

2.Avoid “Premature Generalization”. Create abstractions only when the intent is understood.

3.Do the simplest thing that works, then refactor when necessary.

4.Always make object-behavior transparent to API consumers.

5.Avoid unexpected side-affects when properties, methods, and constructors are invoked.

6.Always separate presentation layer from business logic.

7.Always prefer interfaces over abstract classes.

8.Try to include the design-pattern names such as “Bridge”, “Adapter”, or “Factory” as a suffix to class names where appropriate.

9.Only make members virtual if they are designed and tested for extensibility.

10.Refactor often!

**6. References**

“MSDN: .NET Framework Developer’s Guide: Common Type System”, Microsoft Corporation, 2004, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthecommontypesystem.asp

“MSDN: C# Language Specification v1.5”, Scott Wiltamuth & Anders Hejlsberg, Microsoft Corporation, 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfcsharpspec\_15.asp

“MSDN: Design Guidelines for Class Library Developers”, Microsoft Corporation, 2004, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconNETFrameworkDesignGuidelines.asp

“MSDN: The Well Tempered Exception”, Eric Gunnerson, Microsoft Corporation, 2001 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp08162001.asp

“Applied Microsoft .NET Framework Programming”, Jeffrey Richter, January 23, 2002, 1st ed., Microsoft Press, ISBN: 0735614229

“Which type should I use in C# to represent numbers?”, locabol, February 27, 2007, Luca Bolognese’s Weblog http://blogs.msdn.com/lucabol/archive/2007/02/27/which-type-should-i-use-in-c-to-represent-numbers.aspx