



Coding Standards for .NET

By Lance Hunt

Document Version 1.13a
August 2004

Copyright © Lance Hunt 2004
All Rights Reserved

Published by Lance Hunt
Please submit comments, questions, and feedback to <http://weblogs.asp.net/lhunt/>

License

Usage & Distribution

This work is FREE for educational and non-commercial use. Non-commercial redistribution of the work is permitted so long as all license and all copyright notices are retained and respected. Commercial redistribution of the work or derivative of the work in any form is prohibited unless prior permission is obtained from the copyright holder.

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. 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 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 at any time without notice.

Revision History

Version	Date	Comment
1.0	03/01/2004	Created.
1.01	05/09/2004	Added more Language Usage.
1.02	05/09/2004	Added code examples.
1.03	05/09/2004	Changed formatting and organization.
1.04	05/09/2004	Misc. grammar and syntax changes.
1.05	05/10/2004	Updated Naming Conventions.
1.06	05/10/2004	Misc. adjustments to various rules.
1.07	05/11/2004	Added Bibliography, Resources, and .NET Framework Guidelines.
1.08	05/20/2004	Split .NET guidelines into separate document. Changed overall scope and goals. Restructured sections. Improved Introduction. Added Scope and Terminology sections.
1.09	05/23/2004	Changed style & formatting. Added Quick Summary tables. Added/modified various standards.
1.10	05/25/2004	Modified naming conventions for Internal and Protected identifiers. Added, modified, & removed misc rules. Corrected grammar, code, and some verbiage.
1.11	06/08/2004	Modified formatting. Restructured "Language Usage" section. Corrected language, and added code examples. Consolidated conflicting rules.
1.12	06/30/2004	Modified code commenting and formatting rules. Modified various code examples. Modified Exception Handling and Flow Control sections.
1.13/a	08/17/2004	Modified layout & added License Agreement. Misc. grammar adjustments. Removed rules on foreach vs for pending additional research. Added rules on switch/case statements. <i>Revision(a)</i> subclass Exception not ApplicationException

Table of Contents

1. Introduction	1
1.1 Scope	1
1.2 Document Conventions	1
1.3 Terminology & Definitions	2
1.4 Quick Summary	3
1.4.1 Naming Conventions	3
1.4.2 Coding Style	3
1.4.3 Language Usage	4
2. Naming Conventions	5
2.1 General Guidelines	5
2.2 Name Usage & Syntax	6
3. Coding Style	9
3.1 Formatting	9
3.2 Code Commenting	10
4. Language Usage	11
4.1 General	11
4.2 Variables & Types	11
4.3 Flow Control	12
4.4 Exception Handling	13
4.5 Events, Delegates, & Threading	14
4.6 Object Composition	15
5. Object Model Design	17
6. References	18

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 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.

1.4.1 Naming Conventions

“**c**” = camelCase
 “**P**” = PascalCase
 “**_**” = Prefix with **_**Underscore
 “**x**” = Not Applicable.

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.
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.
Indentation	Use tabs with size of 4.
Comments	Use <code>//</code> or <code>///</code> but not <code>/* ... */</code> and do not flowerbox.
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 <code>int</code> NOT <code>Int32</code>)
Enums	Avoid changing default type.
Generics	Prefer Generic Types over standard or strong-typed classes.
Properties	Never prefix with <code>Get</code> or <code>Set</code> .
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 <code>foreach</code> statement.
Conditionals	Avoid evaluating Boolean conditions against <code>true</code> or <code>false</code> . No embedded assignment. Avoid embedded method invocation.
Exceptions	Do not use exceptions for flow control. Use <code>throw</code> ; not <code>throw e</code> ; when re-throwing. Only catch what you can handle. Use validation to avoid exceptions. Derive from <code>Exception</code> not <code>ApplicationException</code> .
Events	Always check for null before invoking.
Locking	Use <code>lock()</code> not <code>Monitor.Enter()</code> . Do not lock on an object type or <code>"this"</code> . Do lock on private objects.
Dispose() & Close()	Always invoke them if offered, declare where needed.
Finalizers	Avoid. Use the C# Destructors. Do not create <code>Finalize()</code> method.
AssemblyVersion	Increment manually.
ComVisibleAttribute	Set to <code>false</code> 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 namespaces, classes, methods, properties, fields, or parameters that vary only by capitalization.
4. Do not use names that begin with a numeric character.
5. Always choose meaningful and specific names.
6. Always err on the side of verbosity not terseness.
7. Variables and Properties should describe an entity not the type or size.
8. Do not use Hungarian Notation!

Example: `strName` or `iCount`

9. Avoid using abbreviations unless the full name is excessive.
10. Avoid abbreviations longer than 5 characters.
11. Any Abbreviations must be widely known and accepted.
12. Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.
13. Do not use C# reserved words as names.
14. Avoid naming conflicts with existing .NET Framework namespaces, or types.
15. Avoid adding redundant or meaningless prefixes and suffixes to identifiers

Example:

```
// Bad!  
public enum ColorsEnum {...}  
public class CVehicle {...}  
public struct RectangleStruct {...}
```

16. Do not include the parent class name within a property name.
Example: `Customer.Name` NOT `Customer.CustomerName`
17. Try to prefix Boolean variables and properties with “Can”, “Is” or “Has”.
18. Append computational qualifiers to variable names like `Average`, `Count`, `Sum`, `Min`, and `Max` where appropriate.
19. When defining a root namespace, use a Product, Company, or Developer Name as the root.
Example: `LanceHunt.StringUtilities`

2.2 Name Usage & Syntax

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

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

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

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. Avoid declaring multiple **attribute** declarations within a single line. Instead stack each attribute as a separate declaration.

Example:

```
// Bad!
[Attribute1, Attribute2, Attribute3]
public class MyClass
{...}

// Good!
[Attribute1]
[Attribute2]
[Attribute3]
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 written in U.S. English.
- 22. Use `//` or `///` but never `/* ... */`
- 23. Do not “flowerbox” comment blocks.

Example:

```
// *****  
// Comment block  
// *****
```

- 24. Use inline-comments to explain assumptions, known issues, and algorithm insights.
- 25. Do not use inline-comments to explain obvious code. Well written code is self documenting.
- 26. Only insert inline-comments for Bad Code to say “fix this code” – otherwise, rewrite it!
- 27. Include Task-List keyword flags to enable comment-filtering.

Example:

```
// TODO: Place Database Code Here  
// UNDONE: Removed P\Invoke Call due to errors  
// HACK: Temporary fix until able to refactor
```

- 28. Always apply C# comment-blocks (`///`) to `public`, `protected`, and `internal` declarations.
- 29. Only use C# comment-blocks for documenting the API.
- 30. Always include `<summary>` comments. Include `<param>`, `<return>`, and `<exception>` comment sections where applicable.
- 31. Include `<see cref="" />` and `<seealso cref="" />` where possible.
- 32. Always add **CDATA** tags to comments containing code and other embedded markup in order to avoid encoding issues.

Example:

```
/// <example>  
/// Add the following key to the “appSettings” section of your config:  
/// <code><![CDATA[  
///   <configuration>  
///     <appSettings>  
///       <add key=”mySetting” value=”myValue” />  
///     </appSettings>  
///   </configuration>  
/// ]]></code>  
/// </example>
```

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. Afterwards, selectively enable the `ComVisibleAttribute` for individual classes as needed.

Example:

```
[assembly: ComVisible(false)]

[ComVisible(true)]
public MyClass
{...}
```

4. Consider factoring classes with `unsafe` code blocks into a separate assembly.
5. Avoid mutual references between assemblies.

4.2 Variables & Types

6. Try to initialize variables where you declare them.
7. Use the simplest data type, list, or object required. For example, use `int` over `Long` unless you know you need to store 64bit values.
8. 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
```

9. Only declare member variables as `private`. Use properties to provide access to them with `public`, `protected`, or `internal` access modifiers.
10. Avoid specifying a type for an `enum` - use default of `int` unless you have an explicit need for `long`.
11. Avoid using inline numeric literals (magic numbers). Instead, use a `Constant` or `Enum`.
12. Avoid declaring inline string literals. Instead use Constants, Resources, Registry or other data sources.
13. Only declare `constants` for simple types.
14. Declare `readonly` or `static readonly` variables instead of constants for complex types.

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

Example:

```
object dataObject = LoadData();
DataSet ds = dataObject as DataSet;

if(ds != null)
{...}
```

16. Always prefer C# Generic collection types over standard or strong-typed collections.
17. Always explicitly initialize arrays of reference types using a `for` loop.
18. Avoid boxing and unboxing value types.

Example:

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

19. Floating point values should include at least one digit before the decimal place and one after.

Example: `totalPercent = 0.05;`

20. Try to use the “@” prefix for string literals instead of escaped strings.
21. Prefer `String.Format()` or `StringBuilder` over string concatenation.
22. Never concatenate strings inside a loop.
23. Do not compare strings to `String.Empty` or “” to check for empty strings. Instead, compare by using `String.Length == 0`.
24. Avoid hidden string allocations within a loop. Use `String.Compare()` instead.

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

25. Avoid invoking methods within a conditional expression.
26. Avoid creating recursive methods. Use loops or nested loops instead.
27. Avoid using `foreach` to iterate over immutable value-type collections. E.g. String arrays.
28. Do not modify enumerated items within a `foreach` statement.

29. Use the **ternary** conditional operator only for trivial conditions. Avoid complex or compound ternary operations.

Example: `int result = isValid ? 9 : 4;`

30. Avoid evaluating Boolean conditions against `true` or `false`.

Example:

```
// Bad!
if (isValid == true)
{...}

// Good!
if (isValid)
{...}
```

31. Avoid assignment within conditional statements.

Example: `if((i=2)==2) {...}`

32. 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)
{...}
```

33. Avoid explicit Boolean tests in conditionals.

Example:

```
// Bad!
if(IsValid == true)
{...};

// Good!
if(IsValid)
{...}
```

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

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

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

4.4 Exception Handling

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

38. Only `catch` exceptions that you can handle.

39. Never declare an empty `catch` block.

40. Avoid nesting a `try/catch` within a `catch` block.

41. Use exception filters where possible.

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

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

44. If re-throwing an exception, omit the exception argument from the `throw` statement so the original call stack is preserved.

Example:

```
// Bad!
catch(Exception ex)
{
    Log(ex);
    throw ex;
}

// Good!
catch(Exception ex)
{
    Log(ex);
    throw;
}
```

45. Only use the `finally` block to release resources from a `try` statement.
46. 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();
}
```

47. Avoid defining custom exception classes. Use existing exception classes instead.
48. When a custom exception is required;
a. Always derive from `Exception` not `ApplicationException`.
b. Always override the `ToString()` method and `String implicit operator` to provide serialization.
c. Always implement the Exception Constructor Pattern:
`public MyCustomException ();`
`public MyCustomException (string message);`
`public MyCustomException (string message, Exception innerException);`
49. When throwing a new `Exception`, always pass the `innerException` in order to maintain the exception tree & inner call stack.

4.5 Events, Delegates, & Threading

50. Always check Event & Delegate instances for `null` before invoking.
51. Use the default `EventHandler` and `EventArgs` for most simple events.
52. Always derive a custom `EventArgs` class to provide additional data.
53. Use the existing `CancelEventArgs` class to allow the event subscriber to control events.
54. Always use the “`lock`” keyword instead of the `Monitor` type.

- 55. Only lock on a private or private static object.
Example: `lock(myVariable);`
- 56. Avoid locking on a Type.
Example: `lock(typeof(MyClass));`
- 57. Avoid locking on the current object instance.
Example: `lock(this);`

4.6 Object Composition

- 58. Always declare types explicitly within a namespace. Do not use the default "{global}" namespace.
- 59. Avoid declaring methods with more than 7 parameters. Refactor or consider passing a struct or class instead.
- 60. Do not use the "new" keyword to hide members of a derived type.
- 61. Only use the "base" keyword when invoking a base class constructor or base implementation within an override.
- 62. Do not use the `protected` access modifier in `sealed` classes.
- 63. Consider using method overloading instead of the `params` attribute.
- 64. 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))
    {...}
}
```

- 65. Consider overriding `Equals()` on a `struct`.
- 66. Always override the `Equality Operator (==)` when overriding the `Equals()` method.
- 67. Always override the `String Implicit Operator` when overriding the `ToString()` method.
- 68. Always call `Close()` or `Dispose()` on classes that offer it.
- 69. Wrap instantiation of `IDisposable` objects with a "using" statement to ensure that `Dispose()` is automatically called.

Example:

```
using(SqlConnection cn = new SqlConnection(_connectionString))
{...}
```

70. 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);
}
```

71. 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 Design

1. Always prefer delegation over inheritance.
2. Avoid “Premature Generalization”. Create abstractions only when the intent is understood.
3. Do the simplest thing that works, then refactor as time permits.
4. Always make object-behavior transparent to API consumers.
5. Always separate presentation layer from business logic.
6. Always prefer interfaces over abstract classes.
7. Try to append the design-pattern name to class names where appropriate.
8. Make members `virtual` if they are designed and tested for extensibility.
9. Consider using the “`sealed`” keyword to break the inheritance chain.
10. Refactor! Refactor! Refactor!

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>

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