Coding Standard: C#

Abstract

The objective of this coding standard is to have a positive effect on

- Avoidance of errors/bugs, especially the hard-to-find ones.
- Maintainability, by promoting some proven design principles.
- Maintainability, by requiring or recommending a certain unity of style.
- Performance, by dissuading wasteful practices.
- Rules and recommendations are given that promote reliability and maintainability.

Table of Content

1. Introduction

1.1. Objective

This document requires or recommends certain practices for developing programs in the C# language. The objective of this coding standard is to have a positive effect on

- Avoidance of errors/bugs, especially the hard-to-find ones.
- Maintainability, by promoting some proven design principles.
- Maintainability, by requiring or recommending a certain unity of style.
- Performance, by dissuading wasteful practices.

1.2. Scope

This standard pertains to the use of the C# language. With very few exceptions, it does **not** discuss the use of the .NET class libraries. Certain items that deserve attention have been identified, but have **not** been included in this document because treatment in separate documents appears more appropriate. These include items such as:

- Unmanaged code
- COM
- Multi-threading
- Localization (languages, Unicode).
- Remoing WinForms Security

This standard does not include rules or recommendations on how to layout brackets, braces, and code in general.

1.3. Rationale

Reasons to have a coding standard and to comply with it are not given here, except the objectives listed in section 1.1. In this section the origins of the rules and recommendations are given and some explanation why these were chosen.

1.3.1. Sources of inspiration

Many of the rules and recommendations were taken from the MSDN C# Usage Guidelines ([3]). The naming guidelines in that document are identical to those found in Appendix C of the ECMA C# Language Specification ([2]). Naming standards and other style issues are more or less arbitrary, so it seems prudent to follow an existing convention. The naming standard in this document differs from that in the given references only in some miniscule details that will hardly ever occur in practice.

Many other recommendations and a few design patterns were also taken from [3]. The problem with that document is that the guidelines, although quite good, are mostly unsuited for automatic verification.

Some general good practices, most of them concerning Object-Oriented programming, were copied from the PMS-MR C++ Coding Standard ([1]). Some of these are, unsurprisingly, also listed in [3].

Some coding guidelines for Java, a programming language rather similar to C#, have been studied for additional recommendations, but all were too vague and/or too specific to Java to have any impact. Any useful guideline was already present in [1] and/or [3].

The numbering scheme and some of the structure have been copied from [1].

1.3.2. Contrast with C++

A considerable part of a coding standard for C or C++ could be condensed into a single rule, *avoid undefined behavior,* and maybe *shun implementation defined behavior*. Officially C# does not exhibit any of these, barring a few minor, well-defined exceptions. Most examples of undefined behavior in C++ will cause an exception to be thrown in C#. Although this is an improvement on the "*anything might happen"* of C++, it is highly undesirable for post-release software.

1.4. Notational conventions

1.4.1. Rule

A rule should be broken only for compelling reasons where no reasonable alternative can be found. The author
of the violating code shall consult with at least one knowledgeable colleague and a senior designer to review said necessity. A comment in the code explaining the reason for the violation is mandatory.

1.4.2. Recommendation

A recommendation should be followed unless there is good reason to do otherwise. Consultation with a knowledgeable colleague about the validity of the reason is necessary. A comment in the code is recommended.

1.4.3. Checkable

Rules and recommendations in this coding standard are marked checkable if automatic verification of compliance looks feasible[1].

1.4.5. Examples

Please note that the source code formatting in some examples has been chosen for compactness rather than for demonstrating good practice. The use of a certain compact style in some of the examples is considered suitable
for tiny code fragments, but should not be emulated in "real" code.

1.5. Definition of terms and abbreviations

Term	*Description*
GAC	Global Assembly Cache
GC	Garbage Collector
CLR	Common Language Runtime

[1] An acceptable performance rate would be < 3% false positives, < 20 % false negatives

1.6. References

Ref.	*doc. number*	*Author*	*Title*
[1]	XJS-154-1215	Bart van Tongeren	PMS-MR C++ Coding Standard
[2]	ECMA-334	TC39/TG2	C# Language Specification, ed. Dec 2001
[3]	msdn, unnumbered		Design Guidelines for [.NET] Class Library Developers, as

		found on the net[2]
[4]	*SIGPLAN Notices*	Barbara Liskov Data Abstraction and Hierarchy 23,5 (May, 1988).
[5]	Prentice Hall, 1988 Bertrand Meyer	Object Oriented Software Construction
[6]	Philips Medical Systems	C# Coding Standard

2 under http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

2. General rules

2.1 Rule: Every time a recommendation is not followed, this must have a good reason.

Good reasons do not include personal preferences of style.

2.2 Rule: Do not mix code from different providers in one file

In general, third party code will not comply with the coding standard, so do not put such code in the same file as proprietary code.

3. Naming conventions

3.1 Rec: Use US-English for naming identifiers.

3.2 Rule: Use Pascal and Camel casing for naming identifiers.

In Pascal casing the first letter of each word in an identifier is capitalized. For example, `BackColor`.

In Camel casing only the first letter of the second, third, etc. word in a name is capitalized; for example,
`backColor`.

The table below provides the casing for the most common types.

IDENTIFIER	CASE	EXAMPLE
Class	Pascal	`AppDomain`
Enum type	Pascal	`ErrorLevel`
Enum values	Pascal	`FatalError`
Event	Pascal	`ValueChange`
Exception class	Pascal	`WebException`
Field	camel	`listItem`
Const Field	camel	`maximumItems`
Read-only Static Field	camel	`redValue`
Interface	Pascal	`IDisposable`
Method	Pascal	`ToString`
Namespace	Pascal	`System.Drawing`
Parameter	camel	`typeName`
Property	Pascal	`BackColor`

Two-letter abbreviations in Pascal casing have both letters capitalized. In Camel casing this also holds true, except at the start of an identifier where both letters are written in lower case. With respect to capitalization in Pascal and Camel casing, abbreviations with more than two letters are treated as ordinary words. Some examples:

CAMEL CASING	PASCAL CASING
newImage	NewImage
uiEntry	UIEntry
pmsMR	PmsMR

For legacy code exceptions should be allowed:

1. Start ALL fields with _ (underscore) .This part is limited the usage of item 3.7)

2. Properties and methods can have Camel notation (not Pascal)

3.3 Rule: Do not use Hungarian notation or add any other type identification to identifiers.

Use of Hungarian notation is deprecated by companies like Microsoft because it introduces a programming language-dependency and complicates maintenance activities.

Exceptions:
Rule 3.4, 3.12, 3.14, 3.21, 3.20, 3.25, 3.26.

3.4 Rule: Do not prefix member fields.

In general, a method in which it is difficult to distinguish local variables from member fields is too big.

3.5 Rule: Do not use casing to differentiate identifiers.

Some programming languages (e.g. VB.NET) do not support distinguishing identifiers by case, so do not define a type called `A` and `a` in the same context.

This rule applies to namespaces, properties, methods, method parameters, and types. Please note that it **is** allowed to have identifiers that differ only in case in distinct categories, e.g. a property `BackColor` that wraps the field `backColor`.

3.6 Rec: Use abbreviations with care.

Do not contract words in identifiers, but do use well-known abbreviations. For example, do not use `GetWin` instead of `GetWindow`, but **do** use a well-known abbreviation such as UI instead of `UserInterface`.

3.7 Rule: Do not use an underscore in identifiers.

Exception for legacy code.

3.8 Rec: Name an identifier according to its meaning and not its type.

Avoid using language specific terminology in names of identifiers. As an example, suppose you have a number of overloaded methods to write data types into a stream. Do not use definitions like:

```
void Write(double doubleValue);
void Write(long longValue);
```

Instead, use:

```
void Write(double value);
void Write(long value);
```

If it is absolutely required to have a uniquely named method for every data type, use Universal Type Names in the method names. The table below provides the mapping from C# types to Universal types.

sbyte	`Sbyte`
byte	`Byte`
short	`Int16`
ushort	`Uint16`
Int	`Int32`
uint	`Uint32`
long	`Int64`

ulong	`Uint64`
float	`Single`
double	`Double`
bool	`Boolean`
char	`Char`
string	`String`
object	`Object`

Based on the example above, the corresponding reading methods may look like this:

```
double ReadDouble();
long ReadInt64();
```

3.10 Rule: Do not add a suffix to a class or struct name.

Do not add suffixes like `Struct` or `Class` to the name of a `class` or `struct`.

Exceptions:
Rule 3.14 and Rule 3.26.

3.11 Rec: Use a noun or a noun phrase to name a class or struct.

Also, if the class involved is a derived class, it is a good practice to use a compound name. For example, if you have a class named `Button`, deriving from this class may result in a class named `BeveledButton`.

3.12 Rule: Prefix interfaces with the letter I.

All interfaces should be prefixed with the letter `I`. Use a noun (e.g. `IComponent`), noun phrase (e.g. `ICustomAttributeProvider`), or an adjective (e.g. `IPersistable`) to name an interface.

3.13 Rec: Use similar names for the default implementation of an interface.

If you provide a default implementation for a particular interface, use a similar name for the implementing class. Notice that this only applies to classes that **only** implement that interface.

For example, a class implementing the `IComponent` interface could be called `Component` or `DefaultComponent`.

3.14 Rule: Suffix names of attributes with Attribute.

Although this is not required by the C# compiler, this convention is followed by all built-in attributes.

3.15 Rule: Do not add an Enum suffix to an enumeration type.

See also Rule 3.3.

3.16 Rule: Use singular names for enumeration types.

For example, do not name an enumeration type `Protocols` but name it `Protocol` instead. Consider the following example in which only one option is allowed.

```
public enum Protocol
{
    Tcp,
    Udp,
    Http,
    Ftp
}
```

3.17 Rule: Use a plural name for enumerations representing bitfields.

Use a plural name for such enumeration types. The following code snippet is a good example of an enumeration that allows combining multiple options.

```
[Flags]
public Enum SearchOptions
```

```
{
   CaseInsensitive = 0x01,
   WholeWordOnly = 0x02,
   AllDocuments = 0x04,
   Backwards = 0x08,
   AllowWildcards = 0x10
}
```

3.18 Rec: Do not use letters that can be mistaken for digits, and vice versa.

To create obfuscated code, use very short, meaningless names formed from the letters `O`, `o`, `l`, `I` and the digits `0` and `1`. Anyone reading code like

```
bool b001 = (lo == l0) ? (I1 == 11) : (lOl != 101);
```

will marvel at your creativity.

3.19 Rule: Add EventHandler to delegates related to events.

Delegates that are used to define an event handler for an `event` must be suffixed with `EventHandler`. For example, the following declaration is correct for a `Close` event.

```
public delegate CloseEventHandler(object sender, EventArgs arguments)
```

3.20 Rule: Add Callback to delegates related to callback methods.

Delegates that are used to pass a reference to a callback method (so **not** an `event`) must be suffixed with `Callback`. For example:

```
public delegate AsyncIOFinishedCallback(IpcClient client, string message);
```

3.21 Rule: Do not add a Callback or similar suffix to callback methods.

Do not add suffixes like `Callback` or `CB` to indicate that methods are going to be called through a callback delegate. You cannot make assumptions on whether methods will be called through a delegate or not. An end- user may decide to use Asynchronous Delegate Invocation to execute the method.

3.22 Rec: Use a verb for naming an event.

Good examples of events are `Close`, `Minimize`, and `Arrived`. For example, the declaration for the `Close` event may look like this:

```
public event CloseEventHandler Close;
```

3.23 Rule: Do not add an Event suffix (or any other type-related suffix) to the name of an event.

See also Rule 3.3.

3.24 Rule: Use an –ing and –ed form to express pre-events and post- events.

Do not use a pattern like `BeginXxx` and `EndXxx`. If you want to provide distinct events for expressing a point of time before and a point of time after a certain occurrence such as a validation event, do not use a pattern like `BeforeValidation` and `AfterValidation`. Instead, use a `Validat`*`ing`* and `Validat`*`ed`* pattern.

3.25 Rule: Prefix an event handler with On.

It is good practice to prefix the method that is registered as an event handler with `On`. For example, a method that handles the `Closing` event should be named `OnClosing()`.
Exception:
In some situations, you might be faced with multiple classes exposing the same event name. To allow separate event handlers use a more intuitive name for the event handler, as long as it is prefixed with `On`.

3.26 Rule: Suffix exception classes with Exception.

For example: `IpcException`.

3.27 Rule: Do not add code-archive related prefixes to identifiers.

3.28 Rule: Name DLL assemblies after their containing namespace.

To allow storing assemblies in the GAC, their names must be unique. Therefore, use the namespace name as a prefix of the name of the assembly. As an example, consider a group of classes organized under the namespace

`SCC.PmsMR.Platform.OSInterface`. In that case, the assembly generated from those classes will be called `SCC.PmsMR.Platform.OSInterface.dll`.

If multiple assemblies are built from the same namespace, it is allowed to append a unique postfix to the namespace name.

3.29 Rule: Use Pascal casing for naming source files.

Do not use the underscore character and do not use casing to differentiate names of files.

3.30 Rule: Name the source file to the main class

In addition, do not put more than one major class plus its auxiliary classes (such as `EventArgs`-derived classes) in one sourcefile.

4. Comments and embedded documentation

4.1 Rule: Each file shall contain a header block.

The header block must consist of a `#region` block containing the following copyright statement and the name of the file.

```
#region Copyright SoftComputer Consultants 2006
//
// This module is part of the [system]
// Copyright (c) Soft Computer Consultants, Inc.  2006
//    All Rights Reserved
// This document contains unpublished, confidential and proprietary
// information of Soft Computer Consultants, Inc. No disclosure or use of
// any portion of the contents of these materials may be made without the
// express written consent of Soft Computer Consultants, Inc.
//
// Author:       [Name]
// Description:  [Description]
//
// Caution:
#endregion
```

4.2 Rule: Use // for comments.

See Appendix for layout examples.

4.3 Rule: Do not use blocks of //-------- or //********.

4.4 Rule: All comments shall be written in US English.

4.5 Rule: Use XML tags for documenting types and members.

All public and protected types, methods, fields, events, delegates, etc. shall be documented using XML tags. Using these tags will allow IntelliSense to provide useful details while using the types. Also, automatic documentation generation tooling relies on these tags. See A.1 for examples.

Section tags define the different sections within the type documentation.

SECTION TAGS	DESCRIPTION	LOCATION

`<summary>`	Short description	type or member
`<remarks>`	Describes preconditions and other additional information.	type or member
`<param>`	Describes the parameters of a method	method
`<returns>`	Describes the return value of a method	method
`<exception>`	Lists the exceptions that a method or property can throw	method, even or property
`<value>`	Describes the type of the data a property accepts and/or returns	property
`<example>`	Contains examples (code or text) related to a member or a type	type or member
`<seealso>`	Adds an entry to the *See Also* section	type or member
`<overloads>`	Provides a summary for multiple overloads of a method	first method in a overload list.

Inline tags can be used within the section tags.

INLINE TAGS	DESCRIPTION
`<see>`	Creates a hyperlink to another member or type
`<paramref>`	Creates a checked reference to a parameter

Markup tags are used to apply special formatting to a part of a section.

MARKUP TAGS	DESCRIPTION
`<code>`	Changes the indentation policy for code examples
`<c>`	Changes the font to a fixed-wide font (often used with the `<code>` tag)
`<para>`	Creates a new paragraph
`<list>`	Creates a bulleted list, numbered list, or a table.
`<b>`	Bold typeface
`<i>`	Italics typeface

Exception:
In an inheritance hierarchy, do not repeat the documentation but use the `<see>` tag to refer to the base class or interface member.

4.6 Rec: Use #region to group non-public members.

If a class contains a large number of members, attributes, and/or properties, put all non-public members in a region. Preferably, use separate regions to split-up the private, protected, and internal members, and a region to hide all fields. It is also allowed to use the `#region` construct for separating the smaller auxiliary classes from the main class. See also Rule 2.2.

5. Object lifecycle

5.1 Rec: Declare and initialize variables close to where they are used.

5.2 Rec: If possible, initialize variables at the point of declaration.

If you use field initialization then instance fields will be initialized before the instance constructor is called. Likewise, static fields are initialized when the static constructor is called. Notice that the compiler will always initialize any uninitialized reference variable to zero.

Also note that it is perfectly allowed to use the `? :` operator. This is especially relevant in conditional initialization expressions where it is not possible to use selection statements such as `if-else`.

```
enum Color
{
   Red,
   Green
}

bool fatalSituation = IsFatalSituation();
Color backgroundColor = fatalSituation ? Color.Red : Color.Green;
```

5.3 Rule: Declare each variable in a separate declaration statement

5.4 Rule: Use a const field to define constant values.

Making it `const` ensures that memory is allocated for that item only once.

```
private const int maxUsers = 100;
```

Exception
If the value of a constant field must be calculated at run-time (in the static constructor), use a `static read- only` field instead. See also Rec. 5.5.

5.5 Rec: Use a public static read-only field to define predefined object instances.

For example, consider a `Color` class that expresses a certain color internally as red, green, and blue components, and this class has a constructor taking a numeric value, then this class may expose several predefined colors like this.

```
public struct Color
{
  public static readonly Color Red = new Color(0xFF0000);
  public static readonly Color Black = new Color(0x000000);
  public static readonly Color White = new Color(0xFFFFFF);

  public Color(int rgb)
  {
     // implementation
  }
}
```

5.6 Rec: Set a reference field to null to tell the GC that the object is no longer needed.

Setting reference fields to `null` may improve memory usage because the object involved will be unreferenced from that point on, allowing the GC to clean-up the object much earlier. Please note that this recommendation should **not** be followed for a variable that is about to go out of scope.

5.7 Rec: Avoid implementing a destructor.

If a destructor is required, adhere to Rule 5.8 and Rule 5.9.

The use of destructors in C# is demoted since it introduces a severe performance penalty due to way the GC works. It is also a bad design pattern to clean up any resources in the destructor since you cannot predict at which time the destructor is called (in other words, it is non-deterministic).

Notice that C# destructors are not really destructors as in C++. They are just a C# compiler feature to represent CLR Finalizers.

5.8 Rule: If a destructor is needed, also use GC.SuppressFinalize.

If a destructor is needed to verify that a user has called certain cleanup methods such as `Close()` on a `IpcPeer` object, call `GC.SuppressFinalize` in the `Close()` method. This ensures that the destructor is ignored if the user is properly using the class. The following snippet illustrates this pattern.

```
public class IpcPeer
{
    bool connected = false;

    public void Connect()
    {
        // Do some work and then change the state of this object.
        connected = true;
    }
    public void Close()
    {
        // Close the connection, change the state, and instruct the GC
        // not to call the destructor.
        connected = false;
        GC.SuppressFinalize(this);
    }
    ~IpcPeer()
    {
        // If the destructor is called, then Close() was not called.
        if (connected)
        {
            // Warning! User has not called Close(). Notice that you can’t
            // call Close() from here because the objects involved may
            // have already been garbage collected (see Rule 5.9.).
        }
    }
}
```

5.9 Rule: Implement IDisposable if a class uses unmanaged or expensive resources.

If a class uses unmanaged resources such as objects returned by C/C++ DLLs, or expensive resources that must be disposed of as soon as possible, you must implement the `IDisposable` interface to allow class users to explicitly release such resources.

The follow code snippet shows the pattern to use for such scenarios.

```
public class ResourceHolder : IDisposable
{
    ///<summary>
    ///Implementation of the IDisposable interface
    ///</summary>
    public void Dispose()
    {
        // Call internal Dispose(bool)
        Dispose(true);

        // Prevent the destructor from being called
        GC.SuppressFinalize(this);
    }
    ///<summary>
    /// Central method for cleaning up resources
    ///</summary>
    protected virtual void Dispose[4](bool explicit)
    {
        // If explicit is true, then this method was called through the
        // public Dispose()
        if (explicit)
        {
            // Release or cleanup managed resources
        }
        // Always release or cleanup (any) unmanaged resources
    }
    ~ResourceHolder()
    {
         // Since other managed objects are disposed automatically, we
         // should not try to dispose any managed resources (see Rule 5.10.).
         // We therefore pass false to Dispose()
        Dispose(false);
    }
}
```

[4] Please note that this method could have any other name, e.g. `InternalDispose`. It has **no** relation to the parameterless `Dispose()` method of `IDisposable`.

If another class derives from this class, then this class should only override the `Dispose(bool)` method of the base class. It should not implement `IDisposable` itself, nor provide a destructor. The base class’s ‘destructor’ is automatically called.

```
public class DerivedResourceHolder : ResourceHolder
{
   protected override void Dispose(bool explicit)
   {
       if (explicit)
       {
           // Release or cleanup managed resources of this derived
           // class only.
       }
       // Always release or cleanup (any) unmanaged resources.
       // Call Dispose on our base class.
       base.Dispose(explicit);
   }
}
```

5.10 Rule: Do not access any reference type members in the destructor.

When the destructor is called by the GC, it is very possible that some or all of the objects referenced by class members are already garbage collected, so dereferencing those objects may cause exceptions to be thrown.

Only value type members can be accessed (since they live on the stack).

5.11 Rule: Always document when a member returns a copy of a reference type or array

By default, all members that need to return an internal object or an array of objects will return a reference to that object or array. In some cases, it is safer to return a copy of an object or an array of objects. In such case, **always** clearly document this in the specification.

6. Control flow

6.1 Rule: Do not change a loop variable inside a for loop block.

Updating the loop variable within the loop body is generally considered confusing, even more so if the loop variable is modified in more than one place.

6.2 Rec: Update loop variables close to where the loop condition is specified.

This makes understanding the loop much easier.

6.3 Rule: All flow control primitives (if, else, while, for, do, switch) shall be followed by a block, even if it is empty.

Please note that this also avoids possible confusion in statements of the form:

```
if (b1) if (b2) Foo(); else Bar();  // which 'if' goes with the 'else'?
```

6.4 Rule: All switch statements shall have a default label as the last case label.

A comment such as "*no action*" is recommended where this is the explicit intention. If the default case should be unreachable, an assertion to this effect is recommended.

If the default label is always the last one, it is easy to locate.

6.5. Rule: An else sub-statement of an if statement shall not be an if statement without an else part.

The intention of this rule, which applies to `else-if` constructs, is the same as in Rule 6.4. Consider the following example.

```
void Foo(string answer)
{
   if ("no" == answer)
   {
```

```
            Console.WriteLine(“You answered with No”);
         }
         else if (“yes” == answer)
         {
            Console.WriteLine(“You answered with Yes”);
         }
         else
         {
            // This block is required, even though you might not care of any other
            // answers than “yes” and “no”.

         }
```

6.6 Rec: Avoid multiple or conditional return statements.

One entry, one exit is a sound principle and keeps control flow simple. However, if some cases, such as when preconditions are checked, it may be good practice to exit a method immediately when a certain precondition is not met.

6.7 Rec: Do not make explicit comparisons to true or false.

It is usually bad style to compare a `bool`-type expression to `true` or `false`.

Example:

```
while (condition == false)  // wrong; bad style
while (condition != true)   // also wrong
while (((condition == true) == true) == true)   // where do you stop?
while (booleanCondition)    // OK
```

6.8 Rule: Do not access a modified object more than once in an expression.

The evaluation order of sub-expressions within an expression **is** defined in C#, in contrast to C or C++, but such code is hard to understand.

Example:

```
v[i] = ++c;     // right
v[i] = ++i;     // wrong: is v[i] or v[++i] being assigned to?
i = i + 1;      // right
i = ++i + 1;    // wrong and useless; i += 2 would be clearer
```

6.9 Rec: Do not use selection statements (if, switch) instead of a simple assignment or initialization.

Express your intentions directly. For example, rather than

```
bool pos;
if (val > 0)
{
   pos = true;
}
else
{
   pos = false;
}
```

or (slightly better)

```
bool pos = (val > 0) ? true : false;
```

write

```
bool pos;
pos = (val > 0);        // single assignment
```

or even better

```
bool pos = (val > 0);   // initialization
```

7. Object oriented programming

7.1 Rule: Declare all fields (data members) private.

An honored principle, stated in both [1] and [3].

Exceptions to this rule are internal classes.

Exceptions to this rule are `static readonly` fields, which may have any accessibility deemed appropriate. See also Rec. 5.5.

7.2 Rec: Provide a default private constructor if there are only static methods and properties on a class.

Instantiating such a class would be useless.

7.3 Rec: Explicitly define a protected constructor on an abstract base class.

Of course an abstract class cannot be instantiated, so a public constructor should be harmless. However, [3] states:

> *Many compilers will insert a `public` or `protected` constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a `protected` constructor on all abstract classes.*

Dubious reasoning, but harmless. This recommendation is provisional.

7.4 Rec: Selection statements (if-else and switch) should be used when the control flow depends on an object's value; dynamic binding should be used when the control flow depends on the object's type.

This is a general OO principle. Please note that it is usually a design error to write a selection statement that queries the type of an object (keywords `typeof`, `is`).

Exception:

Using a selection statement to determine if some object implements one or more optional interfaces **is** a valid construct though.

7.5 Rule: All variants of an overloaded method shall be used for the same purpose and have similar behavior.

Doing otherwise is against the *Principle of Least Surprise*.

7.6 Rec: If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it.

Using the pattern illustrated below requires a derived class to only override the virtual method. Since all the other methods are implemented by calling the most complete overload, they will automatically use the new implementation provided by the derived class.

```
public class MultipleOverrideDemo
{
    private string someText;

    public MultipleOverrideDemo(string s)
    {
        this.someText = s;
    }

    public int IndexOf(string s)
    {
        return IndexOf(s, 0);
    }

    public int IndexOf(string s, int startIndex)
```

```
        {
            return IndexOf(s, startIndex, someText.Length - startIndex );
        }

        public virtual int IndexOf(string s, int startIndex, int count)
        {
            return someText.IndexOf(s, startIndex, count);
        }
    }
```

An even better approach, **not** required by this coding standard, is to refrain from making `virtual` methods `public`, but to give them `protected`[5] accessibility, changing the sample above into:

```
    public class MultipleOverrideDemo
    {
        // same as above …

        public int IndexOf(string s, int startIndex, int count)
        {
            return InternalIndexOf(s, startIndex, count);
        }

        protected virtual int InternalIndexOf(string s, int startIndex, int count)
        {
            return someText.IndexOf(s, startIndex, count);
        }
    }
```

7.7 Rec: Specify methods using preconditions, postconditions, exceptions; specify classes using invariants.

In other words: attempt to apply *Design by Contract* (see [5]) principles.

You can use `Debug.Assert` to ensure that pre- and post-conditions are only checked in debug builds. In release builds, this method does not result in any code.

7.8 Rec: Use C# to describe preconditions, postconditions, exceptions, and class invariants.

Compilable preconditions etc. are testable.

The exact form (e.g. assertions, special DbC functions such as *require* and *ensure*) is not discussed here. However, a non-testable (text only) precondition is better than a missing one.

7.9. Rule: It shall be possible to use a reference to an object of a derived class wherever a reference to that object's base class object is used.

This rule is known as the *Liskov Substitution Principle*, (see [4]), often abbreviated to *LSP*. Please note that an `interface` is also regarded as a base class in this context.

7.10 Rec: Do not overload any 'modifying' operators on a class type.

In this context the 'modifying' operators are those that have a corresponding assignment operator, i.e. the non- unary versions of `+, -, *, /, %, &, |, ^,` << and >>.

There is very little literature regarding operator overloading in C#. Therefore it is wise to approach this feature with some caution.

Overloading operators on a `struct` type is good practice, since it is a value type. The `class` is a reference type and users will probably expect reference semantics, which are not provided by most operators.

Consider a `class Foo` with an overloaded `operator+(int)`, and thus an impicitly overloaded `operator+=(int)`. If we define the function `AddTwenty` as follows:

```
public static void AddTwenty (Foo f)
{
    f += 20;
}
```

Then this function has **no** net effect:

```
{
    Foo bar = new Foo(5);
    AddTwenty (bar);
    // note that 'bar' is unchanged
    // the Foo object with value 25 is on its way to the GC...
}
```

The exception to this recommendation is a `class` type that has complete value semantics, like `System.String`.

7.11 Rule: Do not modify the value of any of the operands in the implementation of an overloaded operator.

This rule can be found in a non-normative clause of [2], section 17.9.1. Breaking this rule gives counter- intuitive results.

7.12 Rec: If you implement one of operator==(), the Equals method or GetHashCode(), implement all three.

Also override this trio when you implement the `IComparable` interface.

Do consider implementing all relational operators (`!=, <, <=, >, >=`) if you implement any.

If your `Equals` method can throw, this may cause problems if objects of that type are put into a container. Do consider to return `false` for a `null` argument.

The msdn guidelines [3] recommend to return `false` rather than throwing an exception when two incomparable objects, say the proverbial apples and oranges, are compared. Since this approach sacrifices the last remnants of type-safety, this recommendation has been weakened.

7.13 Rec: Use a struct when value semantics are desired.

More precisely, a `struct` should be considered for types that meet any of the following criteria:

Act like primitive types.

Have an instance size under □16 bytes.

Are immutable.

Value semantics are desirable.

Remember that a `struct` cannot be derived from.

7.14 Rule: Allow properties to be set in any order.

Properties should be stateless with respect to other properties, i.e. there should not be an observable difference between first setting property A and then B and its reverse.

7.15 Rec: Use a property rather than a method when the member is a logical data member.

7.16 Rec: Use a method rather than a property when this is more appropriate.

In some cases a method is better than a property:

- The operation is a conversion, such as `Object.ToString`.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the `get` accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important. See Rule 7.14.
- The member is `static` but returns a value that can be changed.
- The member returns a copy of an internal array or other reference type.

Only a `set` accessor would be supplied. Write-only properties tend to be confusing.

7.17 Rule: Do not create a constructor that does not yield a fully initialized object.

Only create constructors that construct objects that are fully initialized. There shall be no need to set additional properties.

7.18 Rule: Always check the result of an as operation.

If you use `as` to obtain a certain interface reference from an object, always ensure that this operation does not return `null`. Failure to do so may cause a `NullReferenceException` at a later stage if the object did not implement that interface.

7.19 Rec: Use explicit interface implementation only to prevent name-clashing or to support optional interfaces.

When you use explicit interface implementation, then the methods implemented by the class involved will not be visible through the class interface. To access those methods, you must first cast the class object to the requested interface.

It is recommended to use explicit interface implementation only:

When you want to prevent name clashing. This can happen when multiple interfaces must be supported which have equally named methods, or when an existing class must support a new interface in which the interface has a member which name clashes with a member of the class.

When you want to support several optional interfaces (e.g. `IEnumerator`, `IComparer`, etc) and you do not want to clutter your class interface with their members.

Consider the following example.

```
public interface IFoo1
{
    void Foo()
}

public interface IFoo2
{
    void Foo()
}
public class FooClass : IFoo1, IFoo2
{
    // This Foo is only accessible by explictly casting to IFoo1
    void IFoo1.Foo() { … }

    // This Foo is only accessible by explictly casting to IFoo2
    void IFoo2.Foo() { … )
}
```

8. Exceptions

8.1 Rule: Only throw exceptions in exceptional situations.

Do not throw exceptions in situations that are normal or expected (e.g. end-of-file). Use return values or status enumerations instead. In general, try to design classes that do not throw exceptions in the normal flow of control. However, **do** throw exceptions that a user is not allowed to catch when a situation occurs that may indicate a design error in the way your class is used.

8.2 Rule: Do not throw exceptions from inside destructors.

When you call an exception from inside a destructor, the CLR will stop executing the destructor, and pass the exception to the base class destructor (if any). If there is no base class, then the destructor is discarded.

8.3 Rec: Only re-throw exceptions when you want to specialize the exception.

Only catch and re-throw exceptions if you want to add additional information and/or change the type of the exception into a more specific exception. In the latter case, set the `InnerException` property of the new exception to the caught exception.

8.4 Rule: List the explicit exceptions a method or property can throw.

Describe the recoverable exceptions using the `<exception>` tag.

Explicit exceptions are the ones that a method or property explicitly throws from its implementation and which users are allowed to catch. Exceptions thrown by .NET framework classes and methods used by this implementation do not have to be listed here.

8.5 Rule: Always log that an exception is thrown.

Logging ensures that if the caller catches your exception and discards it, traces of this exception can be recovered at a later stage.

8.6 Rec: Allow callers to prevent exceptions by providing a method or property that returns the object's state.

For example, consider a communication layer that will throw an `InvalidOperationException` when an attempt is made to call `Send()` when no connection is available. To allow preventing such a situation, provide a property such as `Connected` to allow the caller to determine if a connection is available before attempting an operation.

8.7 Rec: Use standard exceptions.

The .NET framework already provides a set of common exceptions. The table below summarizes the most common exceptions that are available for applications.

EXCEPTION	CONDITION
`ApplicationException`	General application error has occurred that does not fit in the other more specific exception classes.
`IndexOutOfRangeException`	Indexing an array or indexable collection outside its valid range.
`InvalidOperationException`	An action is performed which is not valid considering the object's current state.
`NotSupportedException`	An action is performed which is may be valid in the future, but is not supported.
`ArgumentException`	An incorrect argument is supplied.
`ArgumentNullException`	An null reference is supplied as a method's parameter that does not allow null.
`ArgumentOutOfRangeException`	An argument is not within the required range.

8.8 Rec: Throw informational exceptions.

When you instantiate a new exception, set its `Message` property to a descriptive message that will help the caller to diagnose the problem. For example, if an argument was incorrect, indicate which argument was the cause of the problem. Also mention the name (if available) of the object involved.

Also, if you design a new exception class, note that it is possible to add custom properties that can provide additional details to the caller.

8.9 Rule: Throw the most specific exception possible.

Do not throw a generic exception if a more specific one is available (related to Rec. 8.8.).

8.10 Rule: Only catch the exceptions explicitly mentioned in the documentation.

Moreover, do not catch the base class `Exception` or `ApplicationException`. Exceptions of those classes generally mean that a non-recoverable problem has occurred.

Exception:
On system-level or in a thread-routine, it is allowed to catch the `Exception` class directly, but only when approval by the Senior Designer has been obtained.

8.11 Rule: Derive custom exceptions from ApplicationException.

All exceptions derived from `SystemException` are reserved for usage by the CLR only.

8.12 Rec: Provide common constructors for custom exceptions.

It is advised to provide the three common constructors that all standard exceptions provide as well. These include:

- `XxxException()`
- `XxxException(string message)`
- `XxxException(string message, Exception innerException)`

8.13 Rule: Avoid side-effects when throwing recoverable exceptions.

When you throw a recoverable exception, make sure that the object involved stays in a usable and predictable state. With *usable* it is meant that the caller can catch the exception, take any necessary actions, and continue to use the object again. With *predictable* is meant that the caller can make logical assumptions on the state of the object.

For instance, if during the process of adding a new item to a list, an exception is raised, then the caller may safely assume that the item has not been added, and another attempt to re-add it is possible.

8.14 Rule: Do not throw an exception from inside an exception constructor.

Throwing an exception from inside an exception's constructor will stop the construction of the exception being built, and hence, preventing the exception from getting thrown. The other exception **is** thrown, but this can be confusing to the user of the class or method concerned.

9. Delegates and events

9.1 Rule: Do not make assumptions on the object's state after raising an event.

Prepare for any changes to the current object's state while executing an event handler. The event handler may have called other methods or properties that changed the object's state (e.g. it may have disposed objects referenced through a field).

9.2 Rule: Always document from which thread an event handler is called.

Some classes create a dedicated thread or use the Thread Pool to perform some work, and then raise an event.
The consequence of that is that an event handler is executed from another thread than the main thread. For such an event, the event handler must synchronize (ensure thread-safety) access to shared data (e.g. instance members).

9.3 Rec: Raise events through a protected virtual method.

If a derived class wants to intercept an event, it can override such a virtual method, do its own work, and then decide whether or not to call the base class version. Since the derived class may decide not to call the base class method, ensure that it does not do any work required for the base class to function properly.

Name this method `On`*`EventName`*, where *EventName* should be replaced with the name of the event. Notice that an event handler uses the same naming scheme but has a different signature. The following snippet (most parts left out for brevity) illustrates the difference between the two.

```
///<summary>An example class</summary>
public class Connection
{
    // Event definition
    public event EventHandler Closed;

    // Method that causes the event to occur
    public void Close()
    {
       // Do something and then raise the event
       OnClosed(new EventArgs());

    // Method that raises the Closed event.
    protected OnClosed(EventArgs args)
    {
       Closed(this, args);
    }
}
```

```
///<summary>Main entrypoint.</summary>
public static void Main()
{
   Connection connection = new Connection();
   connection.Closed += new EventHandler(OnClosed);
}
```

```
///<summary>Event handler for the Closed event</summary>
private static void OnClosed(object sender, EventArgs args)
{
    // Implementation left out for brevity.
}
```

9.4 Rule: Use the sender/arguments signature for event handlers.

The goal of this recommendation is to have a consistent signature for all event handlers. In general, the event handler's signature should look like this

```
public delegate void MyEventHandler(object sender, EventArgs arguments)
```

Using the base class as the sender type allows derived classes to reuse the same event handler.

The same applies to the arguments parameter. It is recommended to derive from the .NET Framework's `EventArgs` class and add your own event data. Using such a class prevents cluttering the event handler's signature, allows extending the event data without breaking any existing users, and can accommodate multiple return values (instead of using reference fields). Moreover, all event data should be exposed through properties, because that allows for verification and preventing access to data that is not always valid in all occurrences of a certain event.

9.5 Rec: Implement add/remove accessors if the number of handlers for an event must be limited.

If you implement the `add` and `remove` accessors of an event, then the CLR will call those accessors when an event handler is added or removed. This allows limiting the number of allowed event handlers, or to check for certain preconditions.

9.6 Rec: Consider providing property-changed events.

Consider providing events that are raised when certain properties are changed. Such an event should be named *`Property`*`Changed`, where *Property* should be replaced with the name of the property with which this event is associated.

9.7 Rec: Consider an interface instead of a delegate.

If you provide a method as the target for a delegate, the compiler will only ensure that the method signature matches the delegate's signature.

This means that if you have two classes providing a delegate with the same signature and the same name, and each class has a method as a target for that delegate, it is possible to provide the method of the first class as a target for the delegate in the other class, even though they might not be related at all.

Therefore, it is sometimes better to use interfaces. The compiler will ensure that you cannot accidentally provide a class implementing a certain interface to a method that accepts another interface that happens to have to same name.

10. Various data types

10.1 Rec: Use an enum to strongly type parameters, properties, and return types.

This enhances clarity and type-safety. Try to avoid casting between enumerated types and integral types.

Exception:

In some cases, such as when databases or MIT interfaces that store values as `int`s are involved, using `enum`s will result in an unacceptable amount of casting. In that case, it is better to use a `const int` construction.

10.2 Rule: Use the default type Int32 as the underlying type of an enum unless there is a reason to use Int64.

If the `enum` represents flags and there are currently more than 32 flags, or the `enum` might grow to that many flags in the future, use `Int64`.

Do not use any other underlying type because the Operating System will try to align an `enum` on 32-bit or 64- bit boundaries (depending on the hardware platform). Using a 8-bit or 16-bit type may result in a performance loss.

10.3 Rec: Use the [Flags] attribute on an enum if a bitwise operation is to be performed on the numeric values.

Use an `enum` with the `flags` attribute only if the value can be completely expressed as a set of bit flags. Do not use an `enum` for open sets (such as the operating system version). Use a plural name for such an `enum`, as stated in Rule 3.17.

Usage and effect of this attribute are not mentioned in [2] and not at all clearly in the online documentation, so the benefits of following this recommendation are not obvious.

The intended use appears to be:

```
[Flags]
public enum AccessPrivileges
{
   Read   = 0x1,
   Write  = 0x2,
   Append = 0x4,
   Delete = 0x8,
   All    = Read | Write | Append | Delete
}
```

10.4 Rec: Do not use “magic numbers”.

Do not use literal values, either numeric or strings, in your code other than to define symbolic constants. Use the following pattern to define constants:

```
public class Whatever
{
   public static readonly Color PapayaWhip = new Color(0xFFEFD5);
   public const int MaxNumberOfWheels = 18;
}
```

There are exceptions: the values `0`, `1` and `null` can nearly always be used safely. Very often the values `2` and `-1` are OK as well. Strings intended for logging or tracing are exempt from this rule. Literals are allowed when their meaning is clear from the context, and not subject to future changes.

```
mean = (a + b) / 2;     // okay
```

If the value of one constant depends on the value of another, do attempt to make this explicit in the code, so do **not** write

```
public class SomeSpecialContainer
{
   public const int MaxItems = 32;
   public const int HighWaterMark = 24;     // at 75%
   …
}
```

but rather **do** write

```
public class SomeSpecialContainer
{
   public const int MaxItems = 32;
   public const int HighWaterMark = 3 * MaxItems / 4;   // at 75%
   …
}
```

Please note that an `enum` can often be used for certain types of symbolic constants.

10.5 Rule: Floating point values shall not be compared using either the == or != operators.

Most floating point values have no exact binary representation and have a limited precision.
Exception:

When a floating point variable is explicitly initialized with a value such as 1.0 or 0.0, and then checked for a change at a later stage.

10.6 Rec: Use StringBuilder or String.Format for construction of strings

Strings are immutable, which means that when you concatenate two strings to each other, you effectively create a new string and copy the contents of the other two into it. The more strings are concatenated, the more copying is performed which may result in a dramatic performance loss.

Either create a `StringBuilder` object, or use the `String.Format` method (the latter uses the `StringBuilder` internally). The following example illustrates this.

```
StringBuilder builder = new StringBuilder(“The error ”);

builder.Append(errorMessage); // errorMessage is defined elsewhere
builder.Append(“occurred at ”);
builder.Append(DateTime.Now);

Console.WriteLine(builder.ToString());
```

Alternatively, you can rewrite the previous example as follows:

```
string message = String.Format(“The error {0} occurred at {1}”,
                                errorMessage,
                                DateTime.Now);
```

10.7 Rec: Do not cast types where a loss of precision is possible.

For example, do not cast a `long` (64-bit) to an `int` (32-bit), unless you can guarantee that the value of the `long` is small enough to fit in the `int`.

10.8 Rule: Only implement casts that operate on the complete object.

In other words, do not cast one type to another using a member of the source type. For example, a `Button` class has a `string` property `Name`. It is valid to cast the `Button` to the `Control` (since `Button` **is a** `Control`), but it is not valid to cast the `Button` to a string by returning the value of the `Name` property.

10.9 Rule: *Do not generate a semantically different value with a cast.*

For example, it is appropriate to convert a `Time` or `TimeSpan` into an `Int32`. The `Int32` still represents the time or duration. It does not, however, make sense to convert a file name string such as `c:\mybitmap.gif` into a `Bitmap` object.

11. Coding style

11.1 Rule: *Do not mix coding styles within a group of closely related classes or within a module.*

This coding standard gives you some room in choosing a certain style. Do keep the style consistent within a certain scope. That scope is not rigidly defined here, but is at least as big as a source file.

11.3 Rule: *Write unary, increment, decrement, function call, subscript, and access operators together with their operands.*

This concerns the following operators:

unary:	`& * + - ~ !`
increment and decrement:	`-- ++`
function call and subscript:	`() []`
access:	`.`

It is not allowed to add spaces in between these operators and their operands.

It is not allowed to separate a unary operator from its operand with a newline.

Note: this rule does **not** apply to the **binary** versions of the `& * + -` operators.

example:

```
a = -- b;             // wrong
a = --c;              // right

a = -b - c;           // right
a = (b1 + b2) +
    (c1 - c2) +
    d – e - f;        // also fine: make it as readable as possible
```

11.4 Rule: *Use spaces instead of tabs.*

Different applications interpret tabs differently. Always use spaces instead of tabs. You should change the settings in Visual Studio .NET (or any other editor) for that.

11.5 Rule: *Do not create source lines longer than 80 characters.*

Long lines are hard to read. Many applications, such as printing and difference views, perform poorly with long lines. A maximum line length of 80 characters has proven workable for C and C++ and is what we should strive for. However, C# tends to be more verbose and have deeper nesting compared to C++, so the limit of 80 characters will often cause a statement to be split over multiple lines, thus making it somewhat harder to read.