SOFTWARE ENGINEERING STANDARDS & PRACTICES

# C# Coding Standards

# 9AAD134036

| | |
|---|---|
| Department | GF-IS ADM Applications Performance Excellence (APE) |
| Approver | Giulio Bitella, Global Department Manager |
| Owner | Tomasz Jastrzębski, Global Leader for Software Engineering (SE) |

For the latest distributable version of this and other Software Engineering standards please visit this link to ABB Library.

## WHAT IS THIS?

This document presents set of coding standards, design principles and naming conventions that applies to C# language and .NET framework types. It describes the rules in structure that is easy to read and use so that can be quickly applied by software developers.

## WHAT IS THE PURPOSE OF THIS DOCUMENT?

The goal of the document is to create the main reference and efficiencies across a community of developers. Applying a set of well-defined coding standards will result in code with fewer bugs, better maintainability and ensuring consistency of coding styles between all teams. However, there might be situations when the good code design requires that the below set of rules will be violated. Such as cases should be rare and have clear and compelling reason eventually approved by Technical (Team) Lead Developer.

## TABLE OF CONTENTS

## 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.

### 1. General Guidelines

**C#101** All source code files containing intellectual property owned by ABB must start with the below header containing copyright information. Specified year(s) must state when the content was created.

```
/*
 * Copyright © ABB Inc. 2016-2018
 */
```

**C#102** Avoid putting multiple classes or interfaces in a single file. **Exception**: nested classes.

**C#103** Never declare more than 1 namespace per file.

**C#104** Group extension methods in a class suffixed with `Extensions`. If the name of an extension method conflicts with another member or extension method, you must prefix the call with the class name. Having them in a dedicated class with the `Extensions` suffix improves readability.

**C#105** Append folder-name to namespace for source files within sub-folders.

### 2. Naming Guidelines

**C#201** Always use Camel Case or Upper Camel Case (Pascal Case) names.
**Example**: `MyClass tempInstance = new MyClass(); // Good`

**C#202** Avoid ALL CAPS and all lowercase names. Single lowercase words or letters are acceptable.

**C#203** 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. **Example**:

```
// Bad
private MyClass myInstance;
public MyClass MyInstance;
```

**C#204** Always use grammatically correct US-English for all identifier names.

**C#205** Do not add numeric suffixes to identifier names.

**C#206** Variables and Properties should describe an entity not the type or size.

```
// Bad
private int count1;
private int count2;
```

**C#207** Do not use Hungarian Notation!
**Example**: `strName` or `iCount // Bad`

**C#208** Avoid using abbreviations. Any abbreviations must be widely known and accepted.

**C#209** Capitalize only the first character of the abbreviations.
Example: `SqConnection`, NOT `SQLConnection`
Note this rule applies to namespaces as well.
**Example**:
```
ABB.XYZProject.MySolution.MyClassName // Bad
Abb.XyzProject.MySolution.MyClassName // Good
```

**C#210** Never use underscores in literals besides class member prefix.
**Example**: `RecordId`, not `Record_Id`.

**C#211** Avoid using "of" preposition.
**Example**: `RecordCount`, not `NumberOfRecords`

**C#212** If desired use data kind descriptor at the end.
**Example**: `RegistrationDate`, not `DateRegistered`

**C#213** Prefix interface names with "I", keep interfaces each in a separate file.
**Example**: `IMyInterface // Good`

**C#214** Do not include the parent class name within a property name.
**Example:** `Customer.Name` NOT `Customer.CustomerName`

**C#215** Try to prefix Boolean variables and properties with "Can", "Is" or "Has".

**C#216** Append folder-name to namespace for source files within sub-folders.

### 3. Code Commenting

**C#301** Avoid use inline-comments to explain obvious code. Well written code is self-documenting.

**C#302** All comments and variables should be written in English, be grammatically correct, and contain appropriate punctuation.
Use `//` or `///` but never `/* … */`

**C#303** Always use XML comment-blocks for documenting the API.

**C#304** Do not "flowerbox" comment blocks.
**Example**:
```
// ***********************************
// Comment block
// ***********************************
```

## LANGUAGE USAGE

### 4. General

**C#401** 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)  {…}
```

**#C402** Always use `internal` or `private` access modifiers for types and members, unless you intend to support them as part of a public API.

**C#403** Do not use C# reserved words as literals.

**C#404** Avoid adding redundant or meaningless prefixes and suffixes to identifiers. **Example**:

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

## 5. Variables and Types

**C#501** Always choose the simplest data type, list, or object required.

**C#502** Try to declare member variables as `private` first. Use other access modifiers only when needed.

**C#503** Use `decimal` for variables when operating on financial values.

**C#504** Always prefer C# Generic collection types over standard or strong-typed collections.

**C#505** Avoid boxing and unboxing value types.
**Example**:

```
int count = 1;
object refCount = count;      // implicitly boxed
var newCount = (int)refCount; // explicitly unboxed
```

**C#506** Floating point values should include at least one digit before the decimal place and one after. **Example**: `totalPercent = 0.05;`

**C#507** Never concatenate strings inside a loop. See more in best practices document.

**C#508** Always use `string.IsNullOrEmpty()` or `string.IsNullOrWhitespace()` to check for null or empty strings.

**C#509** Avoid hidden string allocations, especially within a loop.
Use `string.Compare(a, b, false)`
or `string.Equals(a, b, StringComparison.InvariantCultureIgnoreCase)` for case-insensitive comparison.
**Example**: *(ToLower() creates a temp string)*

```
// Bad
var id = -1;
var name = "john";

for (var i = 0; i < customerList.Count; i++)
{
   if(customerList[i].Name.ToLower() == name)
   {
      id = customerList[i].Id;
   }
}
```

```
// Good
var id = -1;
var name = "john";
for (var i = 0; i < customerList.Count; i++)
{
   if(string.Compare(customerList[i].Name, name, true) == 0)
   {
      id = customerList[i].Id;
   }
}
```

**C#510** Use C# 6 string interpolation and `nameof()` operator for increased readability and compile time name check whenever possible and feasible. Example:

```
// Bad
string msg = "File " + fileName + " cannot be read in function
Main.";
// Good
string msg = $"File {fileName} cannot be read in function
{nameof(Main)}.";
```

**C#511** Prefer `string.Format()` or `StringBuilder` over string concatenation for strings build programmatically, i.e. within a loop.

## 6. Flow Control

**C#601** If control block spans multiple lines always use curly brackets.

```
// Bad
if (isValid)
   count++;
else
   count--;

// Good
if (isValid)
{
   count++;
}
else
{
   count--;
}
```

In control blocks curly brackets on the same line are allowed, the same style must be maintained within assembly (application/library)

```
// Good
if (isValid) {
   count++;
} else {
   count--;
}
```

 

```
// Good
if (a == null) throw new ArgumentNullException(nameof(a));
```

**C#602** Use the **ternary** conditional operator only for trivial conditions. Avoid complex or compound ternary operations.
**Example (short):** `var result = isValid ? 9 : 4;`
**Example (long):**

```
var result = isValid
   ? ResultStatus.Success
   : ResultStatus.UnknownError;
```

**C#603** Avoid evaluating Boolean conditions against `true` or `false`.
**Example:**

```
// Bad
if (isValid == true)
{…}
// Good
if (isValid)
{…}
```

**C#604** Besides obvious cases always use `else` clause. If no action within `else` clause is required document the reason.
**Example:**

```
if (count > 10)
{
   return;
}
else
{
   // continue
}
```

**C#605** Never use assignment within conditional statements.
**Example:** `if((i=2)==2) {…} // Bad`

**C#606** Only use `switch/case` statements for simple operations with parallel conditional logic.

## 7. Exceptions

**C#701** Always provide exception message text.

**C#702** Throw the most specific exception that is appropriate. For example, if a method receives a `null` argument, it should throw `ArgumentNullException` instead of its base type `ArgumentException`.

**C#703** Do not use `try/catch` blocks for flow-control.

**C#704** Only `catch` exceptions that you can handle or when you need to perform any action on it (ex. Logging).

**C#705** Never declare an empty `catch` block.

**C#706** 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;
}
```

**C#707** When defining custom exception classes that contain additional properties always:
1. override the `Message` property, `ToString()` method and the implicit operator `string` to include custom property values,
2. modify the deserialization constructor to retrieve custom property values,
3. override the `GetObjectData(…)` method to add custom properties to the serialization collection,
4. consider not preserving the original call stack when it may contain security sensitive information.

## 8. Events and Delegates

**C#801** Always check Event & Delegate instances for `null` before invoking.

**C#802** An event that has no subscribers is null, so before invoking, always make sure that the delegate list represented by the event variable is not null. Furthermore, to prevent conflicting changes from concurrent threads, use a temporary variable to prevent concurrent changes to the delegate.
**Example:**

```
// Good
event EventHandler<NotifyEventArgs> Notify;
void RaiseNotifyEvent(NotifyEventArgs args)
{
   var handlers = Notify;
   if (handlers != null)
   {
      handlers(this, args);
   }
}
```

In C# 6.0 and later simply call: `Notify?.Invoke(this, args);`

**C#803** Use a verb or verb phrase to name an event. For example: `Click, Deleted, Closing, Minimizing, and Arriving`.
**Example:** `public event EventHandler<SearchArgs> Search; // Good`

**C#804** Use -ing and -ed to express pre-events and post-events.
For example, a close event that is raised before a window is closed would be called `Closing` and one that is raised after the window is closed would be called `Closed`. Don't use `Before` or `After` prefixes or suffixes to indicate pre and post events.
Suppose you want to define events related to the deletion process of an object. Avoid defining the Deleting and Deleted events as `BeginDelete` and `EndDelete`. Define those events as follows:
`Deleting`: Occurs just before the object is getting deleted
`Delete`: Occurs when the object needs to be deleted by the event handler.
`Deleted`: Occurs when the object is already deleted.

**C#805** Prefix an event handler with `On`. For example, a method that handles the `Closing` event could be named `OnClosing`.

**C#806** Prefer to derive a custom `EventArgs` class to provide additional data.

**C#807** Avoid passing `null` as the sender argument when raising an event. Identify the sender.

**C#808** Pass `EventArgs.Empty` instead of `null`.
**Exception:** On static events, the sender argument should be `null`.

### 9. Threading

**C#901** Only lock on a private or private static object.
**Example:** `lock(_object); // Good`

**C#902** Never locking on a Type and on "`this`".
**Example:** `lock(typeof(MyClass)); // Bad`

## DESIGN GUIDELINES

### 10. Class Design Guidelines

**C#1001** Use S.O.L.I.D. principles.

**C#1002** Only create a constructor that returns a useful object. There should be no need to set additional properties before the object can be used for whatever purpose it was designed.

**C#1003** Never throw the exception from the constructor besides argument check.

**C#1004** Avoid to refer to derived classes from the base class.

**C#1005** Use Law of Demeter.

**C#1006** Always declare types explicitly within a namespace.

**C#1007** Do not use the default "`global`" namespace.

**C#1008** Always call `Close()` or `Dispose()` on classes that offer it, typically inside `finally` clause. Prefer "`using`" keyword.

**C#1009** If you need to free resources allocated by your type implement `IDisposable` interface. Use good practices – refer to MSDN.
**C#1010** Never throw exception from `Dispose()` method and from finalizers.
**C#1011** Validate public methods arguments.

## REFERENCES


1. Best Coding Practices - 9AAD135446
2. C# Best Coding Practices - 9AAD134037
3. C# Coding Standards - Field Guide - 9AAD134039
4. Java Coding Standards - 9AAD135383
5. SQL Server Coding Standards - 9AAD134842
6. Source Code Management Standards - 9AAD134843


The latest versions of the above standards are available in ABB Library (http://library.abb.com)

## RECOMMENDED READING


1. Albahari, J., & Albahari, B. (2018). *C# 7.0 in a nutshell.* Beijing: O'Reilly.
2. Martin, R. C. (2016). *Clean code: A handbook of agile software craftsmanship.* Upper Saddle River, NJ: Prentice Hall.
3. Skeet, J. (2018). *C# in Depth.* Manning Publications Company.
4. Watson, B. (2018). *Writing High-Performance .NET Code* (2nd ed.). Ben Watson.
5. Aviva Solutions. (n.d.). C# Coding Guidelines. Retrieved from https://csharpcodingguidelines.com


## REVISION HISTORY

| Rev. | Page | Change Description | Author(s) | Date |
|---|---|---|---|---|
| A | all | first version | Tomasz Oleniacz et al. | 2015-02-01 |
| B | all | tech leads review | Tomasz Oleniacz | 2015-03-01 |
| C | all | major review | Wojciech Bartuś et al. | 2017-08-10 |
| D | all | approved | Tomasz Jastrzębski et al. | 2019-04-09 |

# Dev

**Information Systems**

Applications Performance Excellence Department (APE)

Software Engineering Standards & Practices