

---

# Philips Healthcare - C# Coding Standard

## PHILIPS

**Version head**

issued by the CCB Coding Standards Philips Healthcare



# Table of Contents

<b><u>Change History</u></b> .....	<b>1</b>
<b><u>Introduction</u></b> .....	<b>5</b>
<u>Introduction</u> .....	5
<b><u>Comments</u></b> .....	<b>7</b>
<u>Rule 4@101</u> .....	7
<u>Rule 4@105</u> .....	7
<u>Rule 4@106</u> .....	7
<u>Rule 4@111</u> .....	8
<b><u>General</u></b> .....	<b>10</b>
<u>Rule 2@105</u> .....	10
<b><u>Naming</u></b> .....	<b>11</b>
<u>Rule 3@109</u> .....	11
<u>Rule 3@204</u> .....	11
<u>Rule 3@501</u> .....	11
<u>Rule 3@504</u> .....	12
<b><u>Object lifecycle</u></b> .....	<b>13</b>
<u>Rule 5@101</u> .....	13
<u>Rule 5@102</u> .....	13
<u>Rule 5@106</u> .....	13
<u>Rule 5@107</u> .....	14
<u>Rule 5@108</u> .....	14
<u>Rule 5@111</u> .....	15
<u>Rule 5@112</u> .....	15
<u>Rule 5@113</u> .....	16
<u>Rule 5@114</u> .....	18
<u>Rule 5@116</u> .....	18
<u>Rule 5@117</u> .....	18
<u>Rule 5@118</u> .....	19
<u>Rule 5@119</u> .....	19
<u>Rule 5@121</u> .....	19
<u>Rule 5@122</u> .....	20
<b><u>Control flow</u></b> .....	<b>21</b>
<u>Rule 6@101</u> .....	21
<u>Rule 6@102</u> .....	21
<u>Rule 6@105</u> .....	21
<u>Rule 6@109</u> .....	22
<u>Rule 6@112</u> .....	22
<u>Rule 6@115</u> .....	23
<u>Rule 6@119</u> .....	23
<u>Rule 6@120</u> .....	23
<u>Rule 6@121</u> .....	24
<u>Rule 6@191</u> .....	24
<u>Rule 6@201</u> .....	24

# Table of Contents

<b><u>Object oriented</u></b>	<b>26</b>
<u>Rule 7@101</u>	26
<u>Rule 7@102</u>	27
<u>Rule 7@105</u>	27
<u>Rule 7@106</u>	28
<u>Rule 7@107</u>	28
<u>Rule 7@108</u>	28
<u>Rule 7@201</u>	29
<u>Rule 7@301</u>	29
<u>Rule 7@303</u>	29
<u>Rule 7@403</u>	30
<u>Rule 7@404</u>	31
<u>Rule 7@501</u>	32
<u>Rule 7@502</u>	32
<u>Rule 7@503</u>	33
<u>Rule 7@520</u>	33
<u>Rule 7@521</u>	33
<u>Rule 7@530</u>	34
<u>Rule 7@531</u>	34
<u>Rule 7@532</u>	34
<u>Rule 7@533</u>	34
<u>Rule 7@601</u>	35
<u>Rule 7@602</u>	35
<u>Rule 7@603</u>	35
<u>Rule 7@604</u>	36
<u>Rule 7@608</u>	36
<u>Rule 7@609</u>	37
<u>Rule 7@611</u>	37
<u>Rule 7@700</u>	38
<b><u>Exceptions</u></b>	<b>39</b>
<u>Rule 8@102</u>	39
<u>Rule 8@104</u>	40
<u>Rule 8@105</u>	40
<u>Rule 8@107</u>	41
<u>Rule 8@108</u>	41
<u>Rule 8@109</u>	41
<u>Rule 8@110</u>	42
<u>Rule 8@111</u>	42
<u>Rule 8@203</u>	43
<b><u>Delegates and events</u></b>	<b>44</b>
<u>Rule 9@101</u>	44
<u>Rule 9@102</u>	44
<u>Rule 9@108</u>	44
<u>Rule 9@110</u>	45
<u>Rule 9@111</u>	45
<u>Rule 9@112</u>	46
<u>Rule 9@113</u>	46
<u>Rule 9@114</u>	47

# Table of Contents

<b><u>Data types</u></b> .....	<b>48</b>
<u>Rule 10@203</u> .....	48
<u>Rule 10@301</u> .....	49
<u>Rule 10@401</u> .....	49
<u>Rule 10@404</u> .....	50
<u>Rule 10@405</u> .....	50
<u>Rule 10@406</u> .....	51
<u>Rule 10@407</u> .....	51
 <b><u>Coding style</u></b> .....	 <b>52</b>
<u>Rule 11@407</u> .....	52
<u>Rule 11@409</u> .....	53
 <b><u>Performance</u></b> .....	 <b>54</b>
<u>Rule 12@101</u> .....	54
<u>Rule 12@102</u> .....	54
<u>Rule 12@103</u> .....	55
<u>Rule 12@104</u> .....	55
<u>Rule 12@105</u> .....	56
<u>Rule 12@106</u> .....	56
 <b><u>Literature</u></b> .....	 <b>57</b>

# Change History

Revision	Date	Description
5.23	2020-09-04 21:39:30	(Paul Jansen - TIOBE): - RTC ticket 194061: Adjusted severity level of rule 4@101.
5.22	2020-08-28 16:57:24	(Maikel Steneker - TIOBE): - RTC ticket 8242: Added new rule 9@114.
5.21	2020-08-03 16:41:49	(Paul Jansen - TIOBE): - RTC ticket 210846: Set rule 6@119 to "checked".
5.20	2020-06-05 23:55:51	(Paul Jansen - TIOBE): - RTC ticket 207002: Improved links to the Aviva C# Coding Guidelines.
5.19	2020-06-03 13:53:22	(Paul Jansen - TIOBE): - RTC ticket 162719: Improved rule 8@104.
5.18	2020-05-05 15:37:02	(Paul Jansen - TIOBE): - RTC ticket 135894: Removed rules 5@120 and 7@304.
5.17	2020-02-09 13:05:57	(Paul Jansen - TIOBE): - RTC ticket 196631: Added exception to rule 6@112.
5.16	2019-12-01 13:25:52	(Paul Jansen - TIOBE) - RTC ticket 142877: Improved synopsis of rule 5@108.
5.15	2019-08-13 19:46:04	(Paul Jansen - TIOBE) - RTC ticket 162717: Changed rule 7@608.
5.14	2019-08-10 16:17:26	(Paul Jansen - TIOBE) - RTC ticket 156693: Improved description of rule 6@201.
5.13	2019-08-10 15:49:30	(Paul Jansen - TIOBE) - RTC ticket 142625: Improved rule 6@109.
5.12	2019-07-13 18:46:08	(Paul Jansen - TIOBE) - RTC ticket 162720: Removed rule 10@403.
5.11	2019-07-13 18:22:29	(Paul Jansen - TIOBE) - RTC ticket 57283: Improved description of rule 7@102.
5.10	2019-02-24 12:21:56	(Paul Jansen - TIOBE) - RTC ticket 162721: Removed rule 10@501.
5.9	2019-02-02 18:26:55	(Paul Jansen - TIOBE) - RTC ticket 142879: Improved synopsis of rule 5@113.
5.8	2018-12-01 17:31:39	(Paul Jansen - TIOBE) - RTC ticket 142880: Removed rule 6@103.
5.7	2018-09-22 20:16:21	(Paul Jansen - TIOBE) - RTC ticket 142615: Removed rule 3@105.
5.6	2018-08-20 00:09:24	(Paul Jansen - TIOBE) - RTC ticket 142876: Improved the description of rule 4@101.
5.5	2018-08-18 15:33:51	(Paul Jansen - TIOBE) - TiCS ticket 21092: removed non-ascii characters from coding standard.
5.4	2018-07-21 15:49:45	(Paul Jansen - TIOBE) - RTC ticket 142898: Improved the description of rule 6@119.
5.3	2017-01-18 10:40:27	(Bram Stappers - TIOBE) - Ticket 9142: Added example for rule 7@502.
5.2	2016-11-03 11:53:22	(Bram Stappers - TIOBE) - Added rule 5@122 (Avoid empty finalizers).

- 2

		WI-1018 Added new coding rule 5@117
		WI-1036 Set level of coding rule 7@525 from 3 to 8
		WI-1082 Made coding rule 8@204 ¿obsolete¿
		WI-1083 Marked coding rule 6@118 as NOT checked
		WI-1087 Added new coding rule 11@408
		WI-1088 Added new coding rule 11@413
		WI-1097/WI-1096 Adapted coding rule 11@407
		WI-1098 Coding rules 5@111/112&113 should now be in line
		WI-1099 Adapted coding rule 3@102
		WI-1107 Updated coding rule 5@113
		WI-1108 Updated coding rule 5@113
		WI-1110 Added new coding rule 7@533
		WI-1137 Added angle numbers as ¿allowed¿ magic numbers to coding rule 10@301
		WI-1175 Added new coding rule 6@191
		WI-1176 Added new coding rule 6@201
		WI-1641 Adapted coding rule 10@104
3.2	2010-10-07 16:36:14	Added 12@103 performance requirement.
3.1	2010-09-28 16:57:43	IM-TA00007457: Added rule 7@700 (Do not ignore method results) IM-TA00004389: Added rule 3@510 (Use descriptive names for generic type parameters) Added new Performance section IM-TA00007213: Added new rule 12@102. IM-TA00007214: Added new rule 12@101. Also added new performance rule 12@104.
3.0 (Authorized)	2010-03-23 15:55:17	IM-TA00006705 Add reference to 9@103 to rule 3@307 IM-TA00006709 Add INotifyPropertyChanged text to rule 9@106 IM-TA00002697 Relax rule 7@101 (class which contains only fields) IM-TA00006685 C# Coding Standards rule 8@110 should change IM-TA00006707 Make new on rule on casting IM-TA00006706 Rephrase synopsis of 7@102 IM-TA00006831 Remove 3@112 from the C# coding standard
2.5	2009-07-14 14:49:52	IM-TA00006289: Extended rule 7@521 to include the other way around also for value types.
2.4	2009-06-02 10:46:51	Updated levels
2.3	2009-05-12 16:54:40	IM-TA0006022: Exception added to 3@104 IM-TA0006032: 7@401 made obsolete and 7@402 updated IM-TA0006035: 7@610 made obsolete IM-TA0006036: 5@113 extended IM-TA0006037: 7@523 made obsolete
2.2	2009-04-28 13:41:00	Updated checked flag for several rules, which are now checked.
2.1	2009-04-16 11:56:07	Changed severity level range from 1-6 to 1-10 (same as C++ coding standard). Adapted severity levels where possible to same level as C++. Removed reference to "recommendation" in intro.
2.0 (Authorized)	2008-12-23 14:31:28	Authorized by CCB Philips Healthcare
1.11	2008-11-25 16:02:19	Changed recommendations to rules.
1.10	2008-11-25 13:33:39	IM-TA00004683: Extended 8@102 with other locations where we don't expect an exception IM-TA00004383: Removed 1@102 and changed all recommendations in rules IM-TA00004658: Removed 8@201. IM-TA00002912: Extended 10@401
1.9	2008-07-09 08:05:48	Minor fix in 3@101, which caused the PDF output to be incorrect.

- 1.8 2008-07-09 08:02:00 Made 4@110 and 3@502 obsolete (CR's IM-TA00004351 and IM-TA00004350).
- 1.7 2008-06-12 14:02:47 Homepage: Textual changes.  
 3@102: IM-TA00003423, Moved abbreviation part to new rule 3@112.  
 3@109: Remove <code> from namespace. Doesn't make sense and isn't used in the two group specific examples.  
 3@111: remove "name" add end of synopsis.  
 3@112: New rule for IM-TA00003423 and added dicom as an example.  
 3@201: update Synopsis to have "enum" iso "Enum".  
 3@203: IM-TA00002463, changed "Enum" in "enum".  
 3@307: Textual Changes and removed "exception", which wasn't an exception at all.  
 3@402: Textual Changes (reduced MR specificity).  
 3@504: Updated text to formalize it a bit more (better checkable).  
 3@504: IM-TA00003168, added exclusion for Partial classes.  
 3@505: Removed rule, now obsolete.  
 4@103: Removed reference to non-existent appendix.  
 5@105: Removed rule, now obsolete.  
 5@108: IM-TA00003519, adaptations for Shadowing.  
 6@103: IM-TA00002364 (Changed to same text as C++ Coding Standard) and remove one exception to the rule (about switch statements).  
 7@102: Updated text a bit to have 1 exception section with 2 bullets.  
 7@105: IM-TA00003204 (Added exceptions) and removed "dubious" reasoning line. Its confusing.  
 7@303: Textual changes.  
 7@503: IM-TA00003381, made 7@503 obsolete and created new rules: 7@520 till 7@532.  
 7@526: Added rule.  
 7@530/7@532: changed to use HTML gt and lt escape characters.  
 8@107: IM-TA00002462 and rephrased sentence.  
 8@110: removed exception about catching on system-level/thread routine and added SystemException.  
 9@103: Textual changes and added "virtual" to OnClosed() method. Updated first sentence text.  
 9@104: Added note for usage of generic EventHandler.  
 9@108: New rule for IM-TA00003116.  
 9@110: New rule IM-TA00002459.  
 10@203: Adapted text. Found usefulness of using this attribute. Added example.  
 10@401: IM-TA00003380, added >= and <= to rule.
- 1.6 2008-04-25 14:19:35 Changed PMS into Philips Healthcare
- 1.5 2008-04-25 13:26:08 Fixed Philips logo on homepage
- 1.4 2007-08-26 23:22:50 First version in TIOBE's Coding Standard Database.



# Introduction

## 1.1. Objective

This Coding Standard requires 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.

## 1.2. Scope

This standard pertains to the use of the C# language. With few exceptions, it does **not** discuss the use of the .NET class libraries.

The standard is an extension to the Microsoft's Framework Design Guidelines ([\[Microsoft Framework\]](#)).

This standard does not include rules on naming conventions or how to layout code in general. It is recommended to adopt a separate style guide for this.

## 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 are given and some explanation why these were chosen.

### 1.3.1. Sources of inspiration

Some general good practices, most of them concerning Object-Oriented programming, were copied from the Philips Healthcare C++ Coding Standard ([\[C++ Coding Standard\]](#)). Another source from which rules have been adopted is the Aviva C# coding guidelines ([\[Avival\]](#)).

The numbering scheme and some of the structure have been copied from [\[C++ Coding Standard\]](#).

### 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. Applicability

This coding standard applies to all C# code that is part of Philips Healthcare software products or directly supportive to these products. Third party software is constrained by this standard if this software is developed specifically for Philips Healthcare.

## 1.5. Notational conventions

**1.5.1. 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.

**1.5.2. Checkable**

Rules in this coding standard are marked checkable if automatic verification of compliance is enforced by static analyzers.

**1.5.3. 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.

# Comments

## Rules

<u>4@101</u>	Each file shall contain a header block
<u>4@105</u>	All comments shall be written in English
<u>4@106</u>	Use XML tags for documenting types and members
<u>4@111</u>	Don't comment out code

## Rule 4@101

*Synopsis:* Each file shall contain a header block

*Language:* C#

*Level:* 2

*Category:* Comments

### Description

Each file should at least contain the word "Copyright" or "(c)", a reference to the name of the company and a year.

Example:

```
/* Copyright (c) <year> Koninklijke Philips N.V.  
 * All rights are reserved. Reproduction or dissemination  
 * in whole or in part is prohibited without the prior written  
 * consent of the copyright holder.  
 */
```

## Rule 4@105

*Synopsis:* All comments shall be written in English

*Language:* C#

*Level:* 10

*Category:* Comments

### Description

Justification: English is the most common language for programming.

Note that there is no restriction what kind of English is to be used (either UK or US), provided that it is used in a consistent way.

## Rule 4@106

*Synopsis:* Use XML tags for documenting types and members

*Language:* C#

Level: 9

Category: Comments

## Description

All public and protected types, non-trivial methods, fields, events and delegates 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.

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.

## Rule 4@111

*Synopsis:* Don't comment out code

*Language:* C#

*Level:* 3

*Category:* Comments

### **Description**

Never check-in code that is commented-out, but instead use a work item tracking system to keep track of some work to be done. Nobody knows what to do when they encounter a block of commented-out code. Was it temporarily disabled for testing purposes? Was it copied as an example? Should I delete it?

# General

## Rules

<u>2@105</u>	Do not mix code from different providers in one file
--------------	--

## Rule 2@105

*Synopsis:* Do not mix code from different providers in one file

*Language:* C#

*Level:* 6

*Category:* General

### Description

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

Also, avoid mixing code from different Philips departments in one file, e.g., do not mix MR code with PII code. This coding standard does not specify layout rules, so code from both providers may look different.

# Naming

## Rules

<u>3@109</u>	Name namespaces according to a well-defined pattern
<u>3@204</u>	Do not use letters that can be mistaken for digits, and vice versa
<u>3@501</u>	Name DLL assemblies after their containing namespace
<u>3@504</u>	Name the source file to the main class

## Rule 3@109

*Synopsis:* Name namespaces according to a well-defined pattern

*Language:* C#

*Level:* 8

*Category:* Naming

### Description

Namespaces should be written in Pascal casing and should start with the following pattern:

```
<company>.<businessunit>.<technology>.<top-level component>
```

An example is

```
Philips.MR.Cardio.IMM.Common.Logging
```

## Rule 3@204

*Synopsis:* Do not use letters that can be mistaken for digits, and vice versa

*Language:* C#

*Level:* 7

*Category:* Naming

### Description

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 == 10) ? (I1 == 11) : (101 != 101);
```

will wonder what this means.

## Rule 3@501

*Synopsis:* Name DLL assemblies after their containing namespace

*Language:* C#

*Level:* 8

Category: Naming

## Description

To allow storing assemblies in the Global Assembly Cache, 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 `Philips.PmsMR.Platform.OSInterface`. In that case, the assembly generated from those classes will be called `Philips.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.

## Rule 3@504

*Synopsis:* Name the source file to the main class

*Language:* C#

*Level:* 7

*Category:* Naming

## Description

An exception for this rule holds for partial classes. If a partial class is used, then the other files for this class can be named as `MainClass.PostFix.cs`, whereby Postfix is a *meaningful* name which describes the contents and not just `MainClass.2.cs`.

Example: `MyForm.cs` and `MyForm.Designer.cs`.



# Object lifecycle

## Rules

<u>5@101</u>	Declare and initialize variables close to where they are used
<u>5@102</u>	If possible, initialize variables at the point of declaration
<u>5@106</u>	Use a <code>public static readonly</code> field to define predefined object instances
<u>5@107</u>	Set a reference field to <code>null</code> to tell the garbage collector that the object is no longer needed
<u>5@108</u>	Do not re-declare a visible name in a nested scope
<u>5@111</u>	Avoid implementing a finalizer
<u>5@112</u>	Provide a method that will cause the finalizer not to be called
<u>5@113</u>	Implement <code>IDisposable</code> if a class uses unmanaged resources, owns disposable objects or subscribes to other objects
<u>5@114</u>	Do not access any reference type members in the finalizer
<u>5@116</u>	Always document when a member returns a copy of a reference type or array
<u>5@117</u>	Properties, methods and arguments representing strings or collections should never be null
<u>5@118</u>	A virtual method may only be called if an object is fully constructed
<u>5@119</u>	Return an <code>IEnumerable&lt;T&gt;</code> or <code>ICollection&lt;T&gt;</code> instead of a concrete collection class
<u>5@121</u>	Don't use "using" variables outside the scope of the "using" statement
<u>5@122</u>	Avoid empty finalizers

## Rule 5@101

*Synopsis:* Declare and initialize variables close to where they are used

*Language:* C#

*Level:* 7

*Category:* Object lifecycle

## Rule 5@102

*Synopsis:* If possible, initialize variables at the point of declaration

*Language:* C#

*Level:* 7

*Category:* Object lifecycle

### Description

Avoid the C style where all variables have to be defined at the beginning of a block, but rather define and initialize each variable at the point where it is needed.

## Rule 5@106

*Synopsis:* Use a `public static readonly` field to define predefined object instances

*Language:* C#

*Level:* 4

Category: Object lifecycle

## Description

For example, consider a `Color` class/struct 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
    }
}
```

## Rule 5@107

*Synopsis:* Set a reference field to `null` to tell the garbage collector that the object is no longer needed

*Language:* C#

*Level:* 4

*Category:* Object lifecycle

## Description

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

## Rule 5@108

*Synopsis:* Do not re-declare a visible name in a nested scope

*Language:* C#

*Level:* 2

*Category:* Object lifecycle

## Description

Repeating a name that already occurs in an outer scope is seldom intended and may be surprising in maintenance, although the behaviour is well-defined.

```
int foo = something;

if (whatever)
{
    double foo = 12.34;
    double anotherFoo = foo; // Violation.
}
```

**Exception:**

In case a method parameter has the same name as a field then the following construction can be used:

`this.x = x`

```
int foo = something;
?
public void SomeMethod(int foo)
{
    this.foo = foo; // No violation
    int anotherFoo = foo; // However, this again is a violation!
}
```

## Rule 5@111

*Synopsis:* Avoid implementing a finalizer

*Language:* C#

*Level:* 4

*Category:* Object lifecycle

**Description**

If a finalizer is required, adhere to [\[5@112\]](#) and [\[5@113\]](#).

The use of finalizers in C# is demoted since it introduces a severe performance penalty due to way the garbage collector works. Note also that you cannot predict at which time the finalizer is called (in other words, it is non-deterministic).

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

## Rule 5@112

*Synopsis:* Provide a method that will cause the finalizer not to be called

*Language:* C#

*Level:* 3

*Category:* Object lifecycle

**Description**

If a finalizer is needed to verify that a user has called certain cleanup methods, call `GC.SuppressFinalize` in the `Close()` method. This ensures that the finalizer 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 garbage collector
        // not to call the finalizer.
        connected = false;
        GC.SuppressFinalize(this);
    }

    ~IpcPeer()
    {
        // If the finalizer 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@113).
        }
    }
}

```

## Rule 5@113

*Synopsis:* Implement `IDisposable` if a class uses unmanaged resources, owns disposable objects or subscribes to other objects

*Language:* C#

*Level:* 2

*Category:* Object lifecycle

### Description

If a class uses unmanaged resources such as objects returned by C/C++ DLLs, you must implement the `IDisposable` interface to allow class users to explicitly release such resources.

A class should implement the `IDisposable` interface, in case it creates instances of objects that implement the `IDisposable` interfaces and a reference to that instances is kept (note that if the class transfers ownership of the create instance to another object, then it doesn't need to implement `IDisposable`).

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

*Beware also the protection against multiple dispose attempts via the 'already disposed status' member.*

```

public class ResourceHolder : IDisposable
{
    protected bool mIsObjectInstanceAlreadyDisposed = false;

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

        // Prevent the finalizer from being called
        GC.SuppressFinalize(this);
    }
    ///<summary>

```

```

/// Central method for cleaning up resources
///</summary>
protected virtual void Dispose(bool disposing)
{
    if ( ! mIsObjectInstanceAlreadyDisposed)
    {
        // If disposing is true, then this method was called through the
        // public Dispose()
        if (disposing)
        {
            // Release or cleanup managed resources

        }

        // mDatabaseConnection could be null if an exception has occurred
        if (mDatabaseConnection != null) {
            // Always release or cleanup (any) unmanaged resources, E.G.
            if (mDatabaseConnection.State == System.Data.ConnectionStat
            {
                mDatabaseConnection.Close();
                mDatabaseConnection = null;
            }
        }

        mIsObjectInstanceAlreadyDisposed = true;
    }
}
~ResourceHolder()
{
    // Since other managed objects are disposed automatically, we
    // should not try to dispose any managed resources (see Rule 5@114).
    // We therefore pass false to Dispose()
    Dispose(false);
}
}

```

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 finalizer (unless this class also contains 'unmanaged resources').

The derived class only needs to make sure to clean up it's own managed resources and calling the base class' finalizer.

```

public class DerivedResourceHolder : ResourceHolder
{
    protected override void Dispose(bool disposing)
    {
        if ( ! mIsObjectInstanceAlreadyDisposed)
        {
            if (disposing)
            {
                // Release or cleanup managed resources of this derived
                // class only.

            }
            // Always release or cleanup (any) unmanaged resources ...
            // ... by calling Dispose on our base class.
            base.Dispose(disposing);
        }
    }
}

```

## Rule 5@114

*Synopsis:* Do not access any reference type members in the finalizer

*Language:* C#

*Level:* 2

*Category:* Object lifecycle

### Description

When the finalizer is called by the garbage collector, 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 local variables can be accessed (since they live on the stack).

## Rule 5@116

*Synopsis:* Always document when a member returns a copy of a reference type or array

*Language:* C#

*Level:* 5

*Category:* Object lifecycle

### Description

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.

## Rule 5@117

*Synopsis:* Properties, methods and arguments representing strings or collections should never be null

*Language:* C#

*Level:* 4

*Category:* Object lifecycle

### Description

With the introduction of LINQ in C# 3.0, it has become much simpler to write code that manipulates collections and strings (lists, dictionaries, etc.). For this code to work well, and stay simple, it is important that collections and strings are always defined, and not equal to null.

So, as a general rule for C#, we propose the following:

C# members (methods, properties) and arguments that return a collection (such as a list or a dictionary) or string should never return null. Instead of returning null when there is nothing more useful to return, they should always return the empty list, the empty dictionary, the empty string, etc. That way, the caller of such members never has to check for null, and can simply work on the returned collections or strings - whether they are empty or not.

Also, members should document this behavior, i.e., they should document that, indeed, they return an empty collection or string instead of null if nothing more useful can be returned.

## Rule 5@118

*Synopsis:* A virtual method may only be called if an object is fully constructed

*Language:* C#

*Level:* 4

*Category:* Object lifecycle

### Description

Don't call virtual functions from constructor or finalizer. Only call these when the object is in a fully constructed state.

## Rule 5@119

*Synopsis:* Return an `IEnumerable<T>` or `ICollection<T>` instead of a concrete collection class

*Language:* C#

*Level:* 5

*Category:* Object lifecycle

### Description

In general, you don't want callers to be able to change an internal collection, so don't return arrays, lists or other collection classes directly. Instead, return an `IEnumerable<T>`, or, if the caller must be able to determine the count, an `ICollection<T>`.

So

```
public IEnumerable<FooBar> GetRecentItems()
```

is preferred to

```
public List<FooBar> GetRecentItems()
```

Note If you're using .NET 4.5, you can also use `ReadOnlyCollection<T>`, `ReadOnlyList<T>` or `ReadOnlyDictionary<TKey, TValue>`.

## Rule 5@121

*Synopsis:* Don't use "using" variables outside the scope of the "using" statement

*Language:* C#

*Level:* 1

*Category:* Object lifecycle

### Description

The "using" statement is a convenient way to dispose objects. While using this C# feature, one should make sure that the disposed object is not used outside the scope of the "using" statement.

For example:

```
private SsisPackageTester CreateSsisPackageTester(string workbookFile)
{
    using (SsisPackageTester tester = CreateSsisPackageTester(workbookFile))
    {
        tester.SetAssertionVisitors(DefaultAssertionVisitors);
        return tester;
    }
}
```

In this example "tester" is referring to a non-existing object after the "return" statement. Returning a disposed object in this way is an unintended programming error.

## Rule 5@122

*Synopsis:* Avoid empty finalizers

*Language:* C#

*Level:* 3

*Category:* Object lifecycle

### Description

As stated in MSDN, having an empty destructor just causes loss in performance, so should be avoided.

Empty destructors should not be used. When a class contains a destructor, an entry is created in the Finalize queue. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty, this just causes a needless loss of performance.



# Control flow

## Rules

<u>6@101</u>	Do not change a loop variable inside a <code>for</code> loop block
<u>6@102</u>	Update loop variables close to where the loop condition is specified
<u>6@105</u>	All <code>switch</code> statements shall have a <code>default</code> label as the last <code>case</code> label
<u>6@109</u>	Don't return a function at an unexpected location
<u>6@112</u>	Do not make explicit comparisons to <code>true</code> or <code>false</code>
<u>6@115</u>	Do not access a modified object more than once in an expression
<u>6@119</u>	Avoid locking on a public type
<u>6@120</u>	Avoid conditions with double negatives
<u>6@121</u>	Don't use parameters as temporary variables
<u>6@191</u>	Do not dereference null
<u>6@201</u>	The cyclomatic complexity of a method should not exceed its configured maximum.

## Rule 6@101

*Synopsis:* Do not change a loop variable inside a `for` loop block

*Language:* C#

*Level:* 2

*Category:* Control flow

### Description

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. This rule also applies to `foreach` loops.

## Rule 6@102

*Synopsis:* Update loop variables close to where the loop condition is specified

*Language:* C#

*Level:* 4

*Category:* Control flow

### Description

This makes understanding the loop much easier.

## Rule 6@105

*Synopsis:* All `switch` statements shall have a `default` label as the last `case` label

*Language:* C#

*Level:* 2

*Category:* Control flow

## Description

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.

## Rule 6@109

*Synopsis:* Don't return a function at an unexpected location

*Language:* C#

*Level:* 9

*Category:* Control flow

## Description

Minimize the number of returns in each routine. It's harder to understand a routine if, reading it at the bottom, you're unaware of the possibility that it returned somewhere above.

Use a return when it enhances readability. In certain routines, once you know the answer, you want to return it to the calling routine immediately. If the routine is defined in such a way that it doesn't require any cleanup, not returning immediately means that you have to write more code.

## Rule 6@112

*Synopsis:* Do not make explicit comparisons to true or false

*Language:* C#

*Level:* 9

*Category:* Control flow

## Description

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 (conditio) // OK
```

### Exception:

Nullable booleans are allowed to be compared to true or false.

## Rule 6@115

*Synopsis:* Do not access a modified object more than once in an expression

*Language:* C#

*Level:* 5

*Category:* Control flow

### Description

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
```

## Rule 6@119

*Synopsis:* Avoid locking on a public type

*Language:* C#

*Level:* 4

*Category:* Control flow

### Description

In general, avoid locking on a public type, or instances beyond your code's control. The common constructs "lock (this)", "lock (typeof (MyType))", and "lock ( "myLock" )" violate this guideline:

- "lock (this)" is a problem if the instance can be accessed publicly.
- "lock (typeof (MyType))" is a problem if MyType is publicly accessible.
- "lock ( "myLock" )" is a problem because any other code in the process using the same string, will share the same lock.

Best practice is to define a private object to lock on, or a private static object variable to protect data common to all instances.

## Rule 6@120

*Synopsis:* Avoid conditions with double negatives

*Language:* C#

*Level:* 4

*Category:* Control flow

### Description

Although a property like `customer.HasNoOrders` make sense, avoid using it in a negative condition like this:

```
bool hasOrders = !customer.HasNoOrders;
```

Double negatives are more difficult to grasp than simple expressions, and people tend to read over the double negative easily.

## Rule 6@121

*Synopsis:* Don't use parameters as temporary variables

*Language:* C#

*Level:* 5

*Category:* Control flow

### Description

Never use a parameter as a convenient variable for storing temporary state. Even though the type of your temporary variable may be the same, the name usually does not reflect the purpose of the temporary variable.

## Rule 6@191

*Synopsis:* Do not dereference null

*Language:* C#

*Level:* 1

*Category:* Control flow

### Description

All C# code using *reference* types that can be `null` must be 'secured' against inadvertent use of `null` references as follows:

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

## Rule 6@201

*Synopsis:* The cyclomatic complexity of a method should not exceed its configured maximum.

*Language:* C#

*Level:* 4

*Category:* Control flow

### Description

'Cyclomatic complexity' is a software metric (measurement) that directly measures the number of linearly independent paths through a program's source code.

For an often used testing strategy such as 'Basis Path Testing' (McCabe) the number of test cases will equal the cyclomatic complexity. Many studies have found a strong positive correlation between the cyclomatic complexity and the number of defects contained in a module.

As such a 'high' (i.e. bigger than configured max) cyclomatic complexity is an indication of potential defects and/or testability problems.

# Object oriented

## Rules

<a href="#">7@101</a>	Declare all fields (data members) <code>private</code>
<a href="#">7@102</a>	Prevent instantiation of a class if it contains only <code>static</code> members
<a href="#">7@105</a>	Explicitly define a <code>protected</code> constructor on an abstract base class
<a href="#">7@106</a>	Make all types internal by default
<a href="#">7@107</a>	Limit the contents of a source code file to one type
<a href="#">7@108</a>	Use using statements instead of fully qualified type names
<a href="#">7@201</a>	Selection statements ( <code>if-else</code> and <code>switch</code> ) 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
<a href="#">7@301</a>	All variants of an overloaded method shall be used for the same purpose and have similar behavior
<a href="#">7@303</a>	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
<a href="#">7@403</a>	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
<a href="#">7@404</a>	Don't hide inherited members with the <code>new</code> keyword
<a href="#">7@501</a>	Do not overload any 'modifying' operators on a <code>class</code> type
<a href="#">7@502</a>	Do not modify the value of any of the operands in the implementation of an overloaded operator
<a href="#">7@503</a>	If you implement one of <code>operator==( )</code> , the <code>Equals</code> method or <code>GetHashCode( )</code> , implement all three
<a href="#">7@520</a>	Override the <code>GetHashCode</code> method whenever you override the <code>Equals</code> method.
<a href="#">7@521</a>	Override the <code>Equals</code> method whenever you implement the <code>==</code> operator, and make them do the same thing
<a href="#">7@530</a>	Implement operator overloading for the equality ( <code>==</code> ), not equal ( <code>!=</code> ), less than ( <code>&lt;</code> ), and greater than ( <code>&gt;</code> ) operators when you implement <code>Comparable</code>
<a href="#">7@531</a>	Overload the equality operator ( <code>==</code> ), when you overload the addition ( <code>+</code> ) operator and/or subtraction ( <code>-</code> ) operator
<a href="#">7@532</a>	Implement all relational operators ( <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> ) if you implement any
<a href="#">7@533</a>	Do NOT use the <code>Equals</code> method to compare different value types, but use the <i>equality operators</i> instead.
<a href="#">7@601</a>	Allow properties to be set in any order
<a href="#">7@602</a>	Use a property rather than a method when the member is a logical data member
<a href="#">7@603</a>	Use a method rather than a property when this is more appropriate
<a href="#">7@604</a>	Do not create a constructor that does not yield a fully initialized object
<a href="#">7@608</a>	Use pattern matching instead of the <code>"as"</code> keyword
<a href="#">7@609</a>	Use the correct way of casting
<a href="#">7@611</a>	Use generic constraints if applicable
<a href="#">7@700</a>	Do not ignore method results

## Rule 7@101

*Synopsis:* Declare all fields (data members) `private`

*Language:* C#

*Level:* 2

Category: Object oriented

## Description

An honored principle, stated in both [C++ Coding Standard] and [MS Design].

### Exceptions:

- `static readonly` fields and `const` fields, which may have any accessibility deemed appropriate. See also [5@106].
- Classes that only contain fields and no methods/properties.

## Rule 7@102

*Synopsis:* Prevent instantiation of a class if it contains only `static` members

*Language:* C#

*Level:* 5

*Category:* Object oriented

## Description

If a class only contains static members there is no use to instantiate it. The way to prevent instantiation is to mark the class as `static`.

Note that a static class can't be inherited from because there is no use for it (children are only accessible via class names not via instances). So if you have a class with only static methods and it has children, then it is recommended to make a utility class instead.

## Rule 7@105

*Synopsis:* Explicitly define a `protected` constructor on an abstract base class

*Language:* C#

*Level:* 3

*Category:* Object oriented

## Description

Of course an abstract class cannot be instantiated, so a public constructor should be harmless. However, [MS Design] 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.*

## Rule 7@106

*Synopsis:* Make all types internal by default

*Language:* C#

*Level:* 4

*Category:* Object oriented

### Description

To make a more conscious decision on which types to make available to other assemblies first restrict the scope as much as possible. Then carefully decide what to expose as a public type.

Wrong example:

```
class BaseClass
{
    public static int intM = 0;
}
```

Correct example:

```
internal class BaseClass
{
    public static int intM = 0;
}
```

## Rule 7@107

*Synopsis:* Limit the contents of a source code file to one type

*Language:* C#

*Level:* 4

*Category:* Object oriented

### Description

**Exception** Nested types should, for obvious reasons, be part of the same file.

## Rule 7@108

*Synopsis:* Use using statements instead of fully qualified type names

*Language:* C#

*Level:* 5

*Category:* Object oriented

### Description

Limit the usage of fully qualified type names in order to prevent name clashing.

Wrong example:

```
System.Collections.Generic.List<int> list = new System.Collections.Generic.List<int>();
```



Correct example:

```
using System.Collections.Generic;  
List<int> list = new List<int>();
```

If you do need to prevent name clashing, use a using directive to assign an alias:

```
using WebUILabel = System.Web.UI.WebControls.Label;
```

## Rule 7@201

*Synopsis:* 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

*Language:* C#

*Level:* 9

*Category:* Object oriented

### Description

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`). Dynamic binding (e.g. using class inheritance) is a much better choice then.

### Exception:

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

## Rule 7@301

*Synopsis:* All variants of an overloaded method shall be used for the same purpose and have similar behavior

*Language:* C#

*Level:* 3

*Category:* Object oriented

### Description

This is to prevent users to make false assumptions about the semantics of these methods.

## Rule 7@303

*Synopsis:* 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

*Language:* C#

*Level:* 6

Category: Object oriented

## Description

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 );
    }

    // Use virtual for this one.
    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 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);
    }

    // Use virtual for this one.
    protected virtual int InternalIndexOf(string s, int startIndex, int count)
    {
        return someText.IndexOf(s, startIndex, count);
    }
}
```

## Rule 7@403

*Synopsis:* 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

*Language:* C#

*Level:* 3

*Category:* Object oriented

## Description

This rule is known as the *Liskov Substitution Principle*, (see [\[Liskov 88\]](#)), often abbreviated to LSP. Please note that an interface is also regarded as a base class in this context.

## Rule 7@404

*Synopsis:* Don't hide inherited members with the new keyword

*Language:* C#

*Level:* 4

*Category:* Object oriented

## Description

Not only does the new keyword break Polymorphism, one of the most essential object-orientation principles, it also makes subclasses more difficult to understand. Consider the following two classes:

```
public class Book
{
    public virtual void Print()
    {
        Console.WriteLine("Printing Book");
    }
}

public class PocketBook : Book
{
    public new void Print()
    {
        Console.WriteLine("Printing PocketBook");
    }
}
```

This will cause behavior that you would not normally expect from class hierarchies:

```
PocketBook pocketBook = new PocketBook();

pocketBook.Print(); // Will output "Printing PocketBook "
((Book)pocketBook).Print(); // Will output "Printing Book"
```

It should not make a difference whether you call Print through a reference to the base class or through the derived class.

So the correct version would be:

```
public class Book
{
    public virtual void Print()
    {
        Console.WriteLine("Printing Book");
    }
}

public class PocketBook : Book
{
    public override void Print()
    {
        Console.WriteLine("Printing PocketBook");
    }
}
```

```
    }
}
```

## Rule 7@501

*Synopsis:* Do not overload any 'modifying' operators on a `class` type

*Language:* C#

*Level:* 6

*Category:* Object oriented

### Description

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 implicitly 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 rule is a `class` type that has complete value semantics, like `System.String`.

## Rule 7@502

*Synopsis:* Do not modify the value of any of the operands in the implementation of an overloaded operator

*Language:* C#

*Level:* 1

*Category:* Object oriented

### Description

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

```
public static PatientList operator + (PatientList list, Patient p)
{
    list += p;
    return list;
}
```

## Rule 7@503

*Synopsis:* If you implement one of `operator==( )`, the `Equals` method or `GetHashCode( )`, implement all three

*Language:* C#

*Level:* 1

*Category:* Object oriented

### Description

If your `Equals` method can throw an exception, 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 [\[MS Design\]](#) 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.

### Exceptions:

In very rare cases it can be meaningful to override `GetHashCode( )` without implementing the other two.

There is no need to implement the `operator==( )` for *reference* types because a default implementation is available.

## Rule 7@520

*Synopsis:* Override the `GetHashCode` method whenever you override the `Equals` method.

*Language:* C#

*Level:* 1

*Category:* Object oriented

### Description

You must guarantee that for two objects considered equal, according the `Equals` method, the `GetHashCode` method returns the same value.

## Rule 7@521

*Synopsis:* Override the `Equals` method whenever you implement the `==` operator, and make them do the same thing

*Language:* C#

*Level:* 1

*Category:* Object oriented

## Description

This allows infrastructure code such as `Hashtable` and `ArrayList`, which use the `Equals` method, to behave the same way as user code written using the equality operator.

### Note:

For value types, the other way around applies also, i.e., whenever you override the `Equals` method, then also implement the equality operator.

## Rule 7@530

*Synopsis:* Implement operator overloading for the equality (`==`), not equal (`!=`), less than (`<`), and greater than (`>`) operators when you implement `Comparable`

*Language:* C#

*Level:* 3

*Category:* Object oriented

## Rule 7@531

*Synopsis:* Overload the equality operator (`==`), when you overload the addition (`+`) operator and/or subtraction (`-`) operator

*Language:* C#

*Level:* 2

*Category:* Object oriented

## Rule 7@532

*Synopsis:* Implement all relational operators (`<`, `<=`, `>`, `>=`) if you implement any

*Language:* C#

*Level:* 2

*Category:* Object oriented

## Rule 7@533

*Synopsis:* Do NOT use the `Equals` method to compare different value types, but use the *equality operators* instead.

*Language:* C#

*Level:* 3

*Category:* Object oriented

## Description

An example, as given below, is the best way to explain this:

When the types used for `Equals` are different, the `Equals` method returns false regardless of the values:

```
int i = 0;
```

```
uint ui = 0;

if (i == ui)
{
    Console.WriteLine("i == ui: true");
}
else
{
    Console.WriteLine("i == ui: false");
}
if (i.Equals(ui))
{
    Console.WriteLine("i.Equals(ui): true");
}
else
{
    Console.WriteLine("i.Equals(ui): false");
}
```

This code ends with the following output:

```
i == ui: true
i.Equals(ui): false
```

## Rule 7@601

*Synopsis:* Allow properties to be set in any order

*Language:* C#

*Level:* 4

*Category:* Object oriented

### Description

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.

## Rule 7@602

*Synopsis:* Use a property rather than a method when the member is a logical data member

*Language:* C#

*Level:* 9

*Category:* Object oriented

## Rule 7@603

*Synopsis:* Use a method rather than a property when this is more appropriate

*Language:* C#

*Level:* 9

*Category:* Object oriented

## Description

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 [\[7@601\]](#).
- 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.

## Rule 7@604

*Synopsis:* Do not create a constructor that does not yield a fully initialized object

*Language:* C#

*Level:* 2

*Category:* Object oriented

## Description

Only create constructors that construct objects that are fully initialized. There shall be no need to set additional properties. A `private` constructor is exempt from this rule.

## Rule 7@608

*Synopsis:* Use pattern matching instead of the "as" keyword

*Language:* C#

*Level:* 5

*Category:* Object oriented

## Description

Since C# 7 it is possible to perform safe casting by only using the "is" operator. In previous days you had to do something like this:

```
string text = input as string;
if(text != null)
{
    ...
}
```

It is more convenient to write this in the following way:

```
if(input is string text)
{
    ...
}
```



The pattern will not match if input is null and non-nullable types like int are supported as well.

## Rule 7@609

*Synopsis:* Use the correct way of casting

*Language:* C#

*Level:* 7

*Category:* Object oriented

### Description

If a type of an object is determined by design, then use explicit casting. If the type is unknown use the "as" operator and check against null to see if casting succeeded (see [\[7@608\]](#)).

Avoid double casting by first checking the type with the "is" operator and then do the actual casting. This is only possible for reference types.

### Example:

```
ISomeInterface y = null;
if (x is ISomeInterface)    // "is" operator uses a cast.
{
    y = (ISomeInterface)x;  // Here is the second cast.
}

// Better way if by design we know the type.
ISomeInterface y = (ISomeInterface)x;

// Or if we do not know the type use "as"
ISomeInterface y = x as ISomeInterface;
if (y != null)
{
    // Use y.
}
```

## Rule 7@611

*Synopsis:* Use generic constraints if applicable

*Language:* C#

*Level:* 5

*Category:* Object oriented

### Description

Instead of casting to and from the object type in generic types or methods, use where constraints or the as operator to specify the exact characteristics of the generic parameter.

Wrong example:

```
class MyClass<T>
{
    void SomeMethod(T t)
    {
```

```
        object temp = t;  
        SomeClass obj = (SomeClass) temp;  
    }  
}
```

Correct example:

```
class MyClass<T> where T : SomeClass  
{  
    void SomeMethod(T t)  
    {  
        SomeClass obj = t;  
    }  
}
```

## Rule 7@700

*Synopsis:* Do not ignore method results

*Language:* C#

*Level:* 4

*Category:* Object oriented

### Description

There are several situations in which ignoring results is not desired:

- A new object is created but never used. Unnecessary object creation and the associated garbage collection of the unused object degrade performance.
- A method that creates and returns a new string is called and the new string is never used. Strings are immutable and methods such as `String.ToUpper()` returns a new instance of a string instead of modifying the instance of the string in the calling method.
- A COM or P/Invoke method that returns a `HRESULT` or error code that is never used. Ignoring `HRESULT` or error code can lead to unexpected behavior in error conditions or to low-resource conditions.

If the method result is never going to be used, then the design of the method is incorrect. In that case the return type should be set to `void` instead.

# Exceptions

## Rules

<a href="#">8@102</a>	Do not throw exceptions from unexpected locations
<a href="#">8@104</a>	Document the exceptions that are explicitly thrown by a method or a property
<a href="#">8@105</a>	Always log that an exception is thrown
<a href="#">8@107</a>	Use standard exceptions
<a href="#">8@108</a>	Throw informational exceptions
<a href="#">8@109</a>	Throw the most specific exception possible
<a href="#">8@110</a>	Do not silently ignore exceptions
<a href="#">8@111</a>	Throw exceptions rather than returning some kind of status value
<a href="#">8@203</a>	Avoid side-effects when throwing recoverable exceptions

## Rule 8@102

*Synopsis:* Do not throw exceptions from unexpected locations

*Language:* C#

*Level:* 1

*Category:* [Exceptions](#)

### Description

Throwing an exception from some locations are unexpected and can cause problems. For example when you call an exception from inside a finalizer, the CLR will stop executing the finalizer, and pass the exception to the base class finalizer (if any). If there is no base class, then the finalizer is discarded.

Do not throw exceptions from the following locations:

Location	Note
Event accessor methods	The followings exceptions are allowed: <code>System.InvalidOperationException</code> , <code>System.NotSupportedException</code> and <code>System.ArgumentException</code> . This also includes their derivatives.
Equals methods	An Equals method should return true or false. Return false instead of an exception if the arguments do not match.
GetHashCode() methods	GetHashCode() should always return a value, otherwise you lose values in a hash table.
ToString methods	This method is also used by the debugger to display information about objects in a string format. Therefore it should not raise an exception.
Static constructors	A type becomes unusable if an exception is thrown from its static constructor.
Finalizers (finalizers)	Throwing an exception from a finalizer can cause a process to crash.
Dispose methods	Dispose methods are often called in finally clauses as part of cleanup. Also <code>Dispose(false)</code> is called from a finalizer, which in itself should not throw an exception.
Equality Operators ( <code>==</code> , <code>!=</code> )	Like the Equals methods, the operators should always return true or false.

Implicit cast operators	A user is usually unaware that an implicit cast operators is called, therefore throwing an exception from them is unexpected and should not be done.
Exception constructor	Calling a exception constructor is done to throw an exception. If the constructor throws an exception, then this is confusing.

## Rule 8@104

*Synopsis:* Document the exceptions that are explicitly thrown by a method or a property

*Language:* C#

*Level:* 8

*Category:* Exceptions

### Description

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.

Example:

```
/// <exception cref="FileNotFoundException">Thrown when somepath isn't a real file.</exception>
public void MyMethod2()
{
    FileInfo fi = new FileInfo(somepath);
    if (!fi.Exists)
    {
        throw new FileNotFoundException("somepath doesn't exist");
    }
    ...
}
```

This rule is only mandatory for public and internal methods and properties, because the exceptions are in fact part of the API and as such they need to be documented in the same way that everything else about public and internal APIs needs to be documented.

Note also that private methods preferably should not throw exceptions in the first place, since exceptions should be used to handle exceptional situations, and not as a control flow mechanism. There are more graceful and cheaper ways for private methods to deal with this.

## Rule 8@105

*Synopsis:* Always log that an exception is thrown

*Language:* C#

*Level:* 8

*Category:* Exceptions

### Description

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

## Rule 8@107

*Synopsis:* Use standard exceptions

*Language:* C#

*Level:* 3

*Category:* Exceptions

### Description

The following list of exceptions are too generic and should not be raised directly by your code:

- System.Exception
- System.ApplicationException
- Any exception which is reserved for use by the CLR only (check MSDN for this) 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
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	A null reference is supplied as a method's parameter that does not allow null.
ArgumentOutOfRangeException	An argument is not within the required range.

## Rule 8@108

*Synopsis:* Throw informational exceptions

*Language:* C#

*Level:* 6

*Category:* Exceptions

### Description

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.

## Rule 8@109

*Synopsis:* Throw the most specific exception possible

*Language:* C#

*Level:* 6  
*Category:* Exceptions

### Description

Do not throw a generic exception if a more specific one is available (related to [\[8@108\]](#)).

## Rule 8@110

*Synopsis:* Do not silently ignore exceptions  
*Language:* C#  
*Level:* 1  
*Category:* Exceptions

### Description

An empty catch block for all exceptions makes it really hard to find problems. The catch block should at least contain one of the following items:

- Use the exception argument (e.g. in tracing/logging).
- Re-throw the exception (either the same one or as inner exception for a more specific one). The re-throw must be preceded by at least one other statement (logging, cleanup, etc).

Some other rules to obey:

- A simple re-throw only is useless (and can degrade performance) and therefor forbidden.
- Avoid using `throw e;`, since it changes the stack information. A new exception is started from the point of this throw. Just use `throw;` or make a new exception with the caught exception as inner exception.

### Examples:

```
try
{
    ...
} catch (Exception) //
```

## Rule 8@111

*Synopsis:* Throw exceptions rather than returning some kind of status value  
*Language:* C#  
*Level:* 4  
*Category:* Exceptions

### Description

A code base that uses return values for reporting the success or failure tends to have nested if-statements sprinkled all over the code. Quite often, a caller forgets to check the return value anyhow. Structured exception handling has been introduced to allow you to throw exceptions and catch or replace exceptions at a higher layer. In most systems it is quite common to throw exceptions whenever an unexpected situations

occurs.

Wrong example:

```
static bool CopyObject(SampleClass original)
{
    if (original == null)
    {
        return false;
    }
}
```

Correct example:

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new System.ArgumentException("Parameter cannot be null", "original");
    }
}
```

## Rule 8@203

*Synopsis:* Avoid side-effects when throwing recoverable exceptions

*Language:* C#

*Level:* 1

*Category:* Exceptions

### Description

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.

# Delegates and events

## Rules

<u>9@101</u>	Do not make assumptions on the object's state after raising an event
<u>9@102</u>	Always document from which thread an event handler is called
<u>9@108</u>	Use delegate inference instead of explicit delegate instantiation when possible
<u>9@110</u>	Each subscribe must have a corresponding unsubscribe
<u>9@111</u>	Use generic event handler instances
<u>9@112</u>	Prevent passing null values for sender/object to event handler (for instance-based events)
<u>9@113</u>	Always check an event handler delegate for null
<u>9@114</u>	Do not use return values of callbacks in events

## Rule 9@101

*Synopsis:* Do not make assumptions on the object's state after raising an event

*Language:* C#

*Level:* 2

*Category:* Delegates and events

### Description

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).

Typically raising an event should be the last statement in a method.

## Rule 9@102

*Synopsis:* Always document from which thread an event handler is called

*Language:* C#

*Level:* 9

*Category:* Delegates and events

### Description

Some classes create a dedicated thread or use the Thread Pool to perform some work, and then raise an event when the work is done. 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).

## Rule 9@108

*Synopsis:* Use delegate inference instead of explicit delegate instantiation when possible



*Language:* C#  
*Level:* 9  
*Category:* Delegates and events

## Description

Using delegate inference for subscribing to and unsubscribing from event, code can be made much more elegant than the old previous way, which was like:

```
someClass.SomeEvent += new EventHandler(OnHandleSomeEvent);  
private void OnHandleSomeEvent(object sender, EventArgs e)  
{...}
```

This can now be replaced by:

```
someClass.SomeEvent += OnHandleSomeEvent;  
private void OnHandleSomeEvent(object sender, EventArgs e)  
{...}  
}
```

**Note:** this only applies to code written in C# 2.0 and higher.

## Rule 9@110

*Synopsis:* Each subscribe must have a corresponding unsubscribe

*Language:* C#  
*Level:* 2  
*Category:* Delegates and events

## Description

Subscribing to an event gives the object that sends the event, a reference to the subscribed object. If the subscribed object does not unsubscribe once that is not needed, then it will still be called. If for example, the subscribed object is disposed, then the event still is called on that disposed object (which usually is not intended), and also it is not garbage collected. Therefore it is good to ensure that for each subscribe that is done, also an unsubscribe is done, once listening to that event is no longer needed. The `Dispose()` implementation could be used to ensuring that all unsubscribes are done.

## Rule 9@111

*Synopsis:* Use generic event handler instances

*Language:* C#  
*Level:* 5  
*Category:* Delegates and events

## Description

See <http://msdn.microsoft.com/en-us/library/ms182178.aspx> for a complete description and example of this FxCop based rule (CA1003).

Before .NET Framework 2.0, in order to pass custom information to the event handler, a new delegate had to be declared that specified a class that was derived from the `System.EventArgs` class.

This is no longer true in .NET Framework 2.0, which introduced the `System.EventHandler(Of TEventArgs)` delegate.

This generic delegate allows any class that is derived from EventArgs to be used together with the event handler.

A static generic EventHandler will supply a null sender object (there's no object instance!).

An instance EventHandler will supply the object instance that raises the event as sender.

## Rule 9@112

*Synopsis:* Prevent passing null values for sender/object to event handler (for instance-based events)

*Language:* C#

*Level:* 4

*Category:* Delegates and events

### Description

9@104 states "Use the sender/arguments signature for event handlers". This is very good, but if you use that, there are some additional rules you should obey:

- Do not pass null as the event sender parameter when raising a non-static event.  
(Static events do NOT have an (object) instance and thus NO sender.)
- Do not pass null as the event data parameter when raising an event.  
Use **EventArgs.Empty** when no relevant data can/should be sent with the event.

These two come from MSDN (see <http://msdn.microsoft.com/en-us/library/ms229011.aspx>).

## Rule 9@113

*Synopsis:* Always check an event handler delegate for null

*Language:* C#

*Level:* 1

*Category:* Delegates and events

### Description

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.

Wrong example:

```
event EventHandler Notify;
void RaiseNotifyEvent(NotifyEventArgs args)
{
    EventHandler handlers = Notify;
    handlers(this, args);
}
```

Correct example:

```
event EventHandler Notify;
void RaiseNotifyEvent(NotifyEventArgs args)
{
    EventHandler handlers = Notify;
    if (handlers != null)

```

```
    {  
        handlers(this, args);  
    }  
}
```

**Tip** You can prevent the delegate list from being empty altogether. Simply assign an empty delegate like this:

```
event EventHandler Notify = delegate {};
```

## Rule 9@114

*Synopsis:* Do not use return values of callbacks in events

*Language:* C#

*Level:* 2

*Category:* Delegates and events

### Description

Events may have multiple subscribers, in which case their return value has no meaning. For this reason, you should not use a delegate with a return type for events.

```
public delegate bool ThreePpsApplyCallback(XyzDouble p1, XyzDouble p2, XyzDouble p3);  
public event ThreePpsApplyCallback ThreePpsApply; // not allowed  
public ThreePpsApplyCallback ThreePpsApply; // allowed
```

# Data types

## Rules

<a href="#">10@203</a>	Use the [Flags] attribute on an enum if a bitwise operation is to be performed on the numeric values
<a href="#">10@301</a>	Do not use 'magic numbers'
<a href="#">10@401</a>	Floating point values shall not be compared using the == nor the != operators nor the Equals method.
<a href="#">10@404</a>	Only implement casts that operate on the complete object
<a href="#">10@405</a>	Do not generate a semantically different value with a cast
<a href="#">10@406</a>	When using composite formatting, do supply all objects referenced in the format string
<a href="#">10@407</a>	When using composite formatting, do not supply any object unless it is referenced in the format string

## Rule 10@203

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

*Language:* C#

*Level:* 7

*Category:* [Data types](#)

### Description

It is good practice to use the `Flags` attribute for documenting that the enumeration is intended for combinations. Also using this attribute provides an implementation of the `ToString` method, which displays the values in their original names instead of the values.

### Example:

```
FileInfo file = new FileInfo(fileName);
file.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;
Console.WriteLine("file.Attributes = {0}", file.Attributes.ToString());
```

The printed result will be `ReadOnly|Hidden`.

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 [\[3@203\]](#).

### Example:

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

## Rule 10@301

*Synopsis:* Do not use 'magic numbers'

*Language:* C#

*Level:* 7

*Category:* Data types

### Description

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;
}
```

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. For instance, the values 0, 1 and 2 can be used safely. The same holds for mathematical angles 90, 180, 270 and 360, powers of 2 and powers of 10.

```
mean = (a + b) / 2; // okay
WaitMilliseconds(waitTimeInSeconds * 1000); // clear enough
```

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.

## Rule 10@401

*Synopsis:* Floating point values shall not be compared using the == nor the != operators nor the Equals method.

*Language:* C#

Level: 2

Category: Data types

## Description

Most floating point values have no exact binary representation and have a limited precision, which can even decrease by rounding errors, especially when intermediately using integral values. Thus comparing these using `==` resp. `!=` will often not lead to the desired results.

The way to solve this is to define an helper function that takes a range that determines whether two floating point values are the same. E.g.

```
public static bool AlmostEquals(double double1, double double2, double precision)
{
    return (Math.Abs(double1 - double2) <= precision);
}
...
double d1;
double d2;
...
bool equals = AlmostEquals(d1, d2, 0.0000001);
```

It is important to understand that it is wrong to use a fixed constant to compare to because the value of this constant depends on the expected values of the floating points. Big floats need a larger precision than small floats. That's why comparing to constant `Epsilon` is not correct. `Epsilon` might be too small in some cases to perform a correct comparison.

**Exception to the rule** 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.

## Rule 10@404

*Synopsis:* Only implement casts that operate on the complete object

*Language:* C#

Level: 2

Category: Data types

## Description

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.

## Rule 10@405

*Synopsis:* Do not generate a semantically different value with a cast

*Language:* C#

Level: 2

Category: Data types

## Description

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.

## Rule 10@406

*Synopsis:* When using composite formatting, do supply all objects referenced in the format string

*Language:* C#

*Level:* 1

*Category:* Data types

## Description

Composite formatting, e.g. in `String.Format`, uses indexed placeholders that must correspond to elements in the list of values. A runtime exception results if a parameter specifier designates an item outside the bounds of the list of values, and we prefer not to have runtime exceptions.

### Example:

```
Console.WriteLine("The value is {0} and not {1}", i);
```

where the `{1}` specifier designates a missing parameter.

## Rule 10@407

*Synopsis:* When using composite formatting, do not supply any object unless it is referenced in the format string

*Language:* C#

*Level:* 4

*Category:* Data types

## Description

Composite formatting, e.g. in `String.Format`, uses indexed placeholders that must correspond to elements in the list of values. It is not an error to supply objects in that list that are not referenced in the format string, but it very likely a mistake.

### Example:

```
Console.WriteLine("The value is {0} and not {0}", i, j);
```

where the second specifier was probably intended to be `{1}` to refer to `j`.

# Coding style

## Rules

<u>11@407</u>	Write unary, increment, decrement, function call, subscript, and access operators together with their operands; insert spaces around all other operators
<u>11@409</u>	Use spaces instead of tabs

## Rule 11@407

*Synopsis:* Write unary, increment, decrement, function call, subscript, and access operators together with their operands; insert spaces around all other operators

*Language:* C#

*Level:* 10

*Category:* Coding style

### Description

#### OPERATORS & OPERANDS

Operators are operations that are performed, operands are the arguments or expressions of these operations.

E.g. in "int i = (count + 1)":

- there is an assignment operator '=', with operands 'i' and '(count + 1)'
- there is an Add operator '+', with operands 'count' and the literal '1'

Depending on the amount of operands it works on, an operator is called:

- unary -> works on a single operand, e.g. "-1", "++count", "!isClosed" and "sizeof(int)"
- binary -> works on two operands, e.g. "a - b", "(isOnLine && hasImages)" and "(a <= b)"
- ternary -> works on three operands, e.g. the conditional operator '?' as in "((hasImages) ? firstImageIndex : NoImageIndex)"

Next to that an operator can be:

- bitwise -> i.e. applies to the 'bit-pattern' of it's operand(s)
- comparison -> i.e. compares the operands
- logical -> i.e. performs a logical evaluation of it's operands and returns a boolean result (!, &&, ||)
- mathematical -> i.e. performs a mathematical operation (++ , -- , + , - , / , \*)

#### CODING RULES

The following style rules apply for the operators in the table below and their operands:

- 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: the latter rule does **not** apply to the **binary** versions of the '&', '\*', '+' and '-' operators.

For **all other operators** (and their operands) these rules do **NOT** apply !



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

## EXAMPLES

```

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

```

## Rule 11@409

*Synopsis:* Use spaces instead of tabs

*Language:* C#

*Level:* 9

*Category:* Coding style

### Description

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

# Performance

## Rules

<u>12@101</u>	Avoid boxing and unboxing of value types
<u>12@102</u>	Do not use <code>ToLower</code> / <code>ToUpper</code> for case insensitive string comparison
<u>12@103</u>	Consider using <code>Any( )</code> to determine whether an <code>IEnumerable</code> is empty
<u>12@104</u>	Test for empty strings using string length or <code>String.IsNullOrEmpty</code>
<u>12@105</u>	Use the evaluation order of <code>&amp;&amp;</code> (and operator) and <code>  </code> (or operator) to increase performance
<u>12@106</u>	Use <code>List</code> instead of <code>ArrayList</code> especially for value types

## Rule 12@101

*Synopsis:* Avoid boxing and unboxing of value types

*Language:* C#

*Level:* 6

*Category:* Performance

### Description

Boxing and unboxing of values types is an expensive operation and should be avoided.

Some ways to achieve this:

- If there is only one type of object in the `ArrayList`, then replace it by the generic `List` version. Besides the performance gain, the added bonus is type safety. Note: some other collection classes also have generic versions. And when you write your own classes, consider making it a generic class also.
- When putting a value type into a string formatting, use method `ToString( )` on the value type. E.g. instead of writing `String.Format("SomeValue = {0}", x);` write `String.Format("SomeValue = {0}", x.ToString());`

## Rule 12@102

*Synopsis:* Do not use `ToLower`/`ToUpper` for case insensitive string comparison

*Language:* C#

*Level:* 6

*Category:* Performance

### Description

Do not write code like:

```
if (x.ToLower() == y.ToLower()) { } // Compare two strings case insensitive
```

or

```
Hashtable x = new Hashtable();  
if (x.ContainsKey(y.ToUpper())) { } // Compare Hashtable key case insensitive.
```

Using "ToLower()" or "ToUpper()" create a new string which is not required, just use:

```
if (String.Compare(x, y, true) == 0) { }
```

or

```
Hashtable x = new Hashtable(CaseInsensitiveCompare.DefaultInvariant); // Use case insensitive  
if (x.ContainsKey(y)) { }
```

Note: for .NET 1.1 and older use Hashtable constructor with IHashCodeProvider and IComparer.

## Rule 12@103

*Synopsis:* Consider using Any( ) to determine whether an IEnumerable is empty

*Language:* C#

*Level:* 9

*Category:* Performance

### Description

In many cases using Any( ) is more efficient than Count( ). With Count( ) you risk that iterating over the entire collection has a significant impact.

### References

- MS Design enumerable.Count

## Rule 12@104

*Synopsis:* Test for empty strings using string length or String.IsNullOrEmpty

*Language:* C#

*Level:* 6

*Category:* Performance

### Description

When targeting .NET Framework 2.0 or newer, use the IsNullOrEmpty method. Otherwise, use the Length == comparison whenever possible.

Comparing strings using the String.Length property or the String.IsNullOrEmpty method is significantly faster than using Equals. This is because Equals executes significantly more MSIL instructions than either IsNullOrEmpty or the number of instructions executed to retrieve the Length property value and compare it to zero./

You should be aware that Equals and Length == 0 behave differently for null strings. If you try to get the value of the Length property on a null string, the common language runtime throws a System.NullReferenceException. If you perform a comparison between a null string and the empty string, the common language runtime does not throw an exception; the comparison returns false. Testing for null does not significantly affect the relative performance of these two approaches.

## Rule 12@105

*Synopsis:* Use the evaluation order of && (and operator) and || (or operator) to increase performance

*Language:* C#

*Level:* 7

*Category:* Performance

### Description

C# evaluates && and || from left to right and not all arguments need to be evaluated (see below). Use this to put expensive evaluation arguments, such as method calls with an expensive calculation, as the last argument to be evaluated. Example &&: Consider: if (MethodA() && MethodB()) { "do something"; } => if MethodA returns FALSE, MethodB is never called. Use MethodB for expensive evaluation arguments. Example ||: Consider: if (MethodA() || MethodB()) { "do something"; } => if MethodA returns TRUE, MethodB is never called. Use MethodB for expensive evaluation arguments.

## Rule 12@106

*Synopsis:* Use List instead of ArrayList especially for value types

*Language:* C#

*Level:* 7

*Category:* Performance

### Description

For value types, the List is up to 20x faster than the ArrayList. For type string, there is no difference. (This is mainly caused by the fact that the ArrayList, which is a 'left over' from .NET 1.0, is boxing the values and the List isn't.) To prevent potential performance problems in this area the recommendation is to ALWAYS use List.

# Literature

## **C# Lang**

*Title:* C# Language Specification

*Author:* TC39/TG2/TG3

*Year:* 2001

*Publisher:* ecma

*ISBN:* ECMA-334

<http://www.ecma-international.org/publications/standards/Ecma-334.htm>

## **C++ Coding Standard**

*Title:* Philips Healthcare C++ Coding Standard

*Author:* Philips Healthcare CCB Coding Standards

*Year:* 2008

*Publisher:* Philips Healthcare

<http://tics/codingstandards/CPP/viewer>

## **Liskov 88**

*Title:* Data Abstraction and Hierarchy

*Author:* Barbara Liskov

*Year:* 1988

*Publisher:* SIGPLAN Notices, 23,5 (May, 1988)

<http://portal.acm.org/citation.cfm?id=62141>

## **MS Design**

*Title:* Design Guidelines for Developing Class Libraries

*Author:* Microsoft, MSDN

[http://msdn.microsoft.com/en-us/library/ms229042\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms229042(VS.80).aspx)

## **Meyer 88**

*Title:* Object Oriented Software Construction

*Author:* Betrand Meyer

*Year:* 1988

*Publisher:* Prentice Hall

## **Aviva**

*Title:* Aviva C# Coding Guidelines

*Author:* Dennis Doomen

*Year:* 2020

<https://csharpcodingguidelines.com/>

## **Daniel Crabtree**

*Title:* Daniel Crabtree

<https://www.danielcrabtree.com/>

## **MS Programming**

*Title:* C# Programming Guide

*Author:* Microsoft Developer Network

*Publisher:* Microsoft

<https://msdn.microsoft.com/en-us/library/66x5fx1b.aspx>

## **Microsoft Framework**

*Title:* Framework Design Guidelines

*Author:* Krzysztof Cwalina, Brad Abrams

*Year:* 2005

*Publisher:* Addison-Wesley Professional

*ISBN:* 0321246756

**StackOverflow**

*Title:* StackOverflow.com

<https://stackoverflow.com/>