


C# Coding Standards

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

Table of Contents

1.	Introduction	4
1.1	Purpose	4
1.2	Scope	4
2.	C# Coding Standards	5
2.1	C# Source Files	5
2.2	Beginning Comments	5
2.3	Using Statement	5
2.4	Namespace Statement	5
2.5	Class and Interface Declarations	6
2.5.1	Class/Interface Comment	6
2.6	Naming Conventions	6
2.6.1	Method Declaration	7
2.6.2	Variable Declarations	8
2.7	Indentation	9
2.7.1	Line Length	9
2.7.2	Wrapping Lines	9
2.8	Comments	10
2.8.1	Block Comments	10
2.8.2	Single Line Comments	10
2.8.3	Trailing Comments	10
2.8.4	End-Of-Line Comments	11
2.8.5	Documentation Comments	11
2.9	Declarations	11
2.9.1	Number Per Line	11
2.9.2	Initialization	11
2.9.3	Placement	11
2.10	Statements	12
2.10.1	Simple Statements	12
2.10.2	Compound Statements	12
2.10.3	Return Statements	12
2.10.4	if, if-else, if-else-if else Statements	12
2.10.5	for Statements	13
2.10.6	While Statements	14
2.10.7	do-while Statements	14
2.10.8	Switch Statements	14
2.10.9	try-catch Statements	14
2.11	White Space	15
2.11.1	Blank Lines	15
2.11.2	Blank Spaces	15
2.12	Events, Delegates, & Threading	16
3.	References	16

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

Revision History

Version	Author(s)	Description Version	Date Revision	Approved By	Approval Date
1	TIEO	Initial Version	04-Aug-2016	TIEO	04-Aug-2016

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

1. Introduction

1.1 Purpose

This document describes about the coding standards to be followed during application development using Java.

1.2 Scope

This document covers the coding standards pertinent only to C#.Net as per the specification given by Microsoft.

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

2. C# Coding Standards

Why Coding Standards

- ➔ Writing code for others, not for you
- ➔ Greater consistency
- ➔ Easier to understand
- ➔ Easier to maintain
- ➔ Reduces the overall cost of the application
- ➔ Code for people, not for machine

2.1 C# Source Files

- ➔ C# source files should have the extension of .cs
- ➔ Classes should be declared in individual files with the file name matching the class name.
- ➔ Special characters like TAB and page break must be avoided.
 - These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

2.2 Beginning Comments

All source files should begin with comment that lists the class name and copyright notice.

```

/*
 * @(#) Classname
 *
 * Copyright (c) <year>, <name of the company>.
 * All rights reserved.
 *
 * <Copyright notice>
 */

```

2.3 Using Statement

The first non-comment line of most C# source files is a using statement. Remove unused usings by Organize Usings → Remove and Sort

For ex:- using System.Collections;

2.4 Namespace Statement

The namespace statement should be follow with the using statement. The namespace should be the following format.

<top-level domain name>.<company name>.<product name>.<module name>.<layer

Internal/External/Confidential	Page 5 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

name>

For ex:- `Com.Ofs.Frontiersuite.Ormapping.Service`

2.5 Class and Interface Declarations

Class/Interface declarations should be organized in the following manner.

- ➔ Copyright Notice
- ➔ `using` and `namespace` statement
- ➔ Comment
- ➔ `class` or `interface` statement
- ➔ Static variables
- ➔ Instance variables
- ➔ Constructors
- ➔ Methods

2.5.1 Class/Interface Comment

The comment for class/interface should be specified after import statements. It lists the class type, class name, author, product name, module name and created date.

```
/**
 * @author OFS
 * @since Jul 29, 2016
 */
```

2.6 Naming Conventions

The following names should be in Pascal Case i.e., UpperCamelCase

- ➔ File name
- ➔ Namespace
- ➔ Class name
- ➔ Interface name, but prefix with `I`
- ➔ Property
- ➔ Method name
- ➔ Constant except the private one
- ➔ Field name except private fields
- ➔ Static field name except the private one

Internal/External/Confidential	Page 6 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

- ➔ Enum
- ➔ Delegate
- ➔ Event

The following fields should be in Camel Case and prefix with '_' (underscore)

- ➔ Private field
- ➔ Private constant
- ➔ Private Static Field

The following should be in Camel Case

- ➔ Inline variable
- ➔ Method parameters

2.6.1 **Method Declaration**

- ➔ Method name should be verbs
- ➔ Special characters are not allowed
- ➔ No space between a method name and the parenthesis '(' starting its parameter list
- ➔ Open brace '{' appears at the next line of the declaration statement
- ➔ Closing brace '}' starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the '}' should appear immediately after the '{'

```
public class Sample : Object
{
    private void Sample(int i, int j)
    {
        ivar1 = i;
    }

    int emptyMethod() {}
    ...
}
```

- ➔ Abbreviations and acronyms should not be uppercase when used as name.

```
ExportHtmlSource(); // NOT: ExportHTMLSource();
```

- ➔ Generic variables should have the same name as their type.

```
Private void SetTopic(Topic topic) // NOT: void SetTopic(Topic value)
// NOT: void SetTopic(Topic aTopic)
// NOT: void SetTopic(Topic t)
```

Internal/External/Confidential	Page 7 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

db) `private void Connect(Database database) // NOT: void Connect(Database
// NOT: void Connect(Database oracleDB)`

- ➔ The name of the object is implicit, and should be avoided in a method name.

`Line.getLength(); // NOT: Line.GetLineLength();`

- ➔ The term compute can be used in methods where something is computed.

`Value.ComputeAverage();`

- ➔ There are few alternatives to the **is-** prefix that fits better in some situations. These are has, can and should prefixes:

`boolean HasLicense();
boolean CanEvaluate();
boolean ShouldAbort = false;`

- ➔ The term initialize can be used where an object or a concept is established

`printer.InitializeFontset();`

- ➔ The term find can be used in methods where something is looked up

`node.FindShortestPath();`

- ➔ Abbreviations in names should be avoided. But, abbreviations can be used for domain specific phrases like html, cpu, pdf, etc.,

`ComputeAverage(); // NOT: CompAvg();
ActionEvent event; // NOT: ActionEvent e;
catch (Exception exception) // NOT: catch (Exception e)`

2.6.2 Variable Declarations

- ➔ Always choose the simplest data type, list, or object required. Always use the built-in C# data type aliases, not the .NET common type system (CTS).

`short NOT System.Int16`

`int NOT System.Int32`

`long NOT System.Int64`

`string NOT System.String`

- ➔ Static and Instance variables are declared in the following order in the class. First public, then protected, then namespace level (no access modifier), and then the private
- ➔ Private variables should be begin with underscore '_' and the access modifiers (set and get methods) of that variable should be begin with uppercase. It is best practice to identify the private variables while developing.
- ➔ Declare readonly or static readonly variables instead of constants for complex types.

Internal/External/Confidential	Page 8 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

- ➔ Final variables must be all uppercase using underscore '_' to separate words.

```
final int NUMBER_OF_HOURS_IN_A_DAY = 24
```

- ➔ All names should be written in English

Variables with a large scope should have long name, variables with a small scope can have short names. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for character *c* and *d*.

- ➔ Access modifier (*get* and *set*) should be used where an attribute is accessed directly

- ➔ *Is* prefix is used for boolean variables and methods

```
IsSet, IsVisible, IsFinished, IsFound, IsOpen
```

- ➔ Plural form should be used on names representing a collection of objects

```
List<String> points;
int[] values;
```

- ➔ Negated boolean variable names must be avoided.

```
bool IsError; // NOT: IsNoError
bool IsFound; // NOT: IsNotFound
```

- ➔ The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 can be considered declared as named constants or enum instead.

```
private static final int TEAM_SIZE = 11;
```

```
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players =
new Player[11];
```

- ➔ Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0; // NOT: double total = 0;
double sum = (a + b) * 10.0;
```

- ➔ Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5; // NOT: double total = .5;
```

2.7 Indentation


2.7.1 Line Length

The maximum length of each line is 132 characters.

2.7.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:-

Internal/External/Confidential	Page 9 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

- ➔ Break after a comma ','
- ➔ Break before an operator '+,-,{'
- ➔ Align the new line with the beginning of the expression to increase the indent (4 spaces) on the previous line
- ➔ Break with complete expression

For ex:-

```
SomeMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);

int a = SomeMethod1(longExpression1,
                   SomeMethod2(longExpression2,
                               longExpression3));

lLongName1 = longName2 * (longName3 + longName4 - longName5)
               + 4 * longname6; // prefer high level breaks

longName1 = longName2 * (longName3 + longName4
                       - longName5) + 4 * longname6; // Avoid lower-level breaks
```

2.8 Comments

2.8.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

```
/*
 * Here is a block comment.
 */
```

2.8.2 Single Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line.

```
if (condition)
{
    /* Handle the condition. */
    ...
}
```

2.8.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in

Internal/External/Confidential	Page 10 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

a chunk of code, they should all be indented to the same tab setting.

```

if (a == 2)
{
    return TRUE;           /* special case */
}
else
{
    return isPrime(a);     /* works only for odd a */
}

```

2.8.4 End-Of-Line Comments

The '/' comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments

```

if (foo > 1)
{
    // Do a double-flip.
    ...
}

```

2.8.5 Documentation Comments

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters '/*...*/' with one comment per class, interface, or member.

```

/**
 * The Example class provides ...
 */
public class Example { ...

```

2.9 Declarations

2.9.1 Number Per Line

One declaration per line is recommended since it encourages commenting.

```

int level; // indentation level
int size;  // size of table
int level, size; (Use it, if do not want to put comment about the variables)

```

2.9.2 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

2.9.3 Placement


Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}"). Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```

Private void MyMethod()

```

Internal/External/Confidential	Page 11 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

```

{
    int int1 = 0;           // beginning of method block

    if (condition)
    {
        int int2 = 0;      // beginning of "if" block
        ...
    }
}

```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block

```

int count;
...
MyMethod()
{
    if (condition)
    {
        int count = 0;      // avoid it
        ...
    }
    ...
}

```

2.10 Statements

2.10.1 Simple Statements

Each line should contain at most one statement

```

argv++;           // Correct
argc--;           // Correct
argv++; argc--;  // AVOID!

```

2.10.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces '{ statements }'.

The enclosed statements should be indented one more level than the compound statement.

The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as an `if-else` or `for` statement.

2.10.3 Return Statements

A return statement with a value should not use parentheses.

```

return;
return a;

```

2.10.4 if, if-else, if-else-if else Statements

The `if-else` class of statements should have the following form:-

Internal/External/Confidential	Page 12 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

```

if (condition)
{
    statements;
}

if (condition)
{
    statements;
}
else
{
    statements;
}

if (condition)
{
    statements;
}
else if (condition)
{
    statements;
}
else
{
    statements;
}

```

Use ternary operator for simple condition.

```
int biggestNo = (a > b) ? a : b;
```

instead of

```

int biggestNo = 0;
if (a > b)
{
    biggestNo = a;
}
else
{
    biggestNo = b;
}

```

➔ The conditional should be put on a separate line.

```

if (isDone)          // NOT: if (isDone) doCleanup();
    doCleanup();

```

2.10.5 for Statements

A for statement should have the following form:

```
for (initialization; condition; update)
```

Internal/External/Confidential	Page 13 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

```
{
    statements;
}
```

2.10.6 While Statements

A while statement should have the following form:

```
while (condition)
{
    statements;
}
```

2.10.7 do-while Statements

A do-while statement should have the following form:

```
do
{
    statements;
} while (condition);
```

2.10.8 Switch Statements

A switch statement should have the following form:

```
switch (condition)
{
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

2.10.9 try-catch Statements

A try-catch statement should have the following format:

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

Internal/External/Confidential	Page 14 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

```

try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
finally
{
    statements;
}

```

2.11 White Space

2.11.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- ➔ Between sections of a source file
- ➔ Between class and interface definitions

One blank line should always be used in the following circumstances:

- ➔ Between methods
- ➔ Between the local variables in a method and its first statement
- ➔ Before a block or single-line comment
- ➔ Between logical sections inside a method to improve readability

2.11.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- ➔ A keyword followed by a parenthesis should be separated by a space. Example:

```

while (true)
{
    ...
}

```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- ➔ A blank space should appear after commas in argument lists.
- ➔ All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and

Internal/External/Confidential	Page 15 of 16
Controlled copy	Do not duplicate

Rev.No	1.2	C# Coding Standards	
Rev.Dt	14-Sep-09		
UIC	OFS/PRC/CDR-1/T01		

decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);
```

```
while (d++ = s++)
{
    n++;
}
```

```
PrintSize("size is " + foo + "\n");
```

➔ The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

➔ Casts should be followed by a blank space. Examples:

```
MyMethod((int) aNum, (string) x);
```

2.12 Events, Delegates, & Threading

- ➔ Always check Event & Delegate instances for `null` before invoking.
- ➔ Use the default `EventHandler` and `EventArgs` for most simple events.
- ➔ Always derive a custom `EventArgs` class to provide additional data.
- ➔ Use the existing `CancelEventArgs` class to allow the event subscriber to control events.
- ➔ Always use the "lock" keyword instead of the `Monitor` type.
- ➔ Only lock on a private or private static object.

```
lock(myVariable);
```

➔ Avoid locking on a Type.

```
lock(typeof(MyClass));
```

➔ Avoid locking on the current object instance.

```
lock(this);
```

3. References

- ➔ <http://msdn.microsoft.com/en-us/library/vstudio/ff926074.aspx>
- ➔ <http://www.codeproject.com/Articles/8971/C-Coding-Standards-and-Best-Programming-Practices>
- ➔ <http://www.dofactory.com/reference/csharp-coding-standards.aspx>
- ➔ <http://se.inf.ethz.ch/old/teaching/ss2007/251-0290-00/project/CSharpCodingStandards.pdf>