Coding Standards

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.  - Martin Fowler

Last update: 28 April 2015

Version: V1.0

Contents

[1. Basic knowledge of Coding Standards 3](#_Toc405203222)

[2. C# Coding Standards 3](#_Toc405203223)

[2.1 Naming Guidelines 3](#_Toc405203224)

[2.2 Member Design Guidelines 3](#_Toc405203225)

[2.3 Miscellaneous Design & Layout Conventions 3](#_Toc405203226)

[2.4 Object Design Guidelines 3](#_Toc405203227)

# Basic knowledge of Coding Standards

A comprehensive coding standard is essential for a successful product delivery. It is very imperative for MyTE to have a coding standard to according to when we are writing code. The standard helps in enforcing best practices and avoiding pitfalls, and makes knowledge dissemination across the team easier. Not following any standard is like going with a temporary solution (which might lead to a permanent problem) and, as you will see, it takes less effort to keep in mind a few simple measures than to do haphazard coding. All you have to do is study good standards once and keep them in the back of your head.

Although some might see coding guidelines as undesired overhead or something that limits creativity, this approach has already proven its value for many years. Why? Well, because not every developer

* is aware that code is generally read 10 times more than it is changed;
* is aware of the potential pitfalls of certain constructions in C#;
* is introduced into certain conventions when using the .NET Framework such as IDisposable or the deferred execution nature of LINQ;
* is aware of the impact of using (or neglecting to use) particular solutions on aspects like security, performance, multi-language support, etc.
* knows that not every developer is as capable in understanding an elegant, but abstract, solution as the original developer;

# C# Coding Standards

## Naming Guidelines

Do  **Use US-English.**

All type members, parameters and variables should be named using words from the American English language.

* Choose easily readable, preferably grammatically correct names. For example, Horizontal Alignment is more readable than Alignment Horizontal.
* Favor readability over brevity. The property name CanScrollHorizontally is better than ScrollableX (an obscure reference to the X-axis).
* Avoid using names that conflict with keywords of widely used programming languages.

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do   **We must make sure the name is meaningful and descriptive.**

* Avoid single-character variable names, such as i or t. Use index or temp instead.
* Avoid using Hungarian notation for public or protected members.
* Avoid abbreviating words (such as num instead of number).
* Avoid fully qualified type names. Use the using statement instead.

**Why:** we need take some efforts to maintain it and easy to make mistakes. We have SpecFlow and Nunit test code instead.

Do  **Use proper casing for language elements.**

|  |  |  |
| --- | --- | --- |
| **Language element** | **Casing** | **Example** |
| Class, Struct | Pascal | AppDomain |
| Interface | Pascal | IBusinessService |
| Enumeration type | Pascal | ErrorLevel |
| Enumeration values | Pascal | FatalError |
| Event | Pascal | Click |
| Private field | Pascal | \_selectedItem |
| Private method | Pascal | \_IsExpired() |
| Protected field | Pascal | MainPanel |
| Const field | Upper | ACTIVE\_EMPLOYMENT\_STATUS |
| Const local variable | Camel | maximumItems |
| Read-only static field | Pascal | RedValue |
| Local Variable | Camel | listOfValues |
| Method | Pascal | ToString |
| Namespace | Pascal | System.Drawing |
| Parameter | Camel | typeName |
| Type Parameter | Pascal | TView |
| Property | Pascal | BackColor |
| Abbreviations In MyTE | Upper | WBS/BD/SAP/FFP/PPA…. |

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do  **Name methods using verb-object pair.**

1. // It is good.
2. public bool ValidateEmailAddress()
3. {
4. ….
5. }
6. public bool IsDelegatorSearchBarEnabled()
7. {
8. ….
9. }
10. public TimeReport GetTimeReportStatusByPeopleKey()
11. {
12. ….
13. }
14. // It is not good
15. public bool ValidationEmailAddress()
16. {
17. ….
18. }
19. public TimeReport TimeReportStatus()
20. {
21. ….
22. }

**Why:** easy to understand the meaning of the function and know it’s an action.

Do  **Use a verb or verb phrase to name an event. Use -ing and -ed to express pre-events and post-events. Prefix an event handler with On.**

1. // It is good.
2. public event SelectedRowEventHandler OnFilterChanged;
3. public event PropertyChangedEventHandler OnPropertyChanged;
4. public event PropertyChangingEventHandler OnPropertyChanging;
5. // It is not good
6. public event RemoveHandler RemoveSelected;
7. public event PropertyChangedEventHandler PropertyChanged;
8. public event PropertyChangedEventHandler OnPropertyChange;

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do  **Group extension methods in** **a class suffixed with Extensions.**

1. // It is good put extension method in a class suffixed with Extensions
2. public static class TransactionExtensions
3. {
4. public static void ConfigureForReadCommitted(this TransactionTemplate template)
5. {
6. if (template == null)
7. {
8. return;
9. }
10. ...
11. }
12. }

**Why:** It is a good coding habit, and it is easy to know about the class by its name.

Do  **Suffix custom attribute classes with Attribute. Suffix custom attribute classes with** **Attribute.**

1. // It is good.
2. public sealed class SerializableAttribute : Attribute
3. {
4. internal static Attribute GetCustomAttribute(Type type)
5. {
6. return (type.Attributes & TypeAttributes.Serializable) == TypeAttributes.Serializable ? new SerializableAttribute() : null;
7. }
8. internal static bool IsDefined(Type type)
9. {
10. return type.IsSerializable;
11. }
12. public SerializableAttribute()
13. {
14. ….
15. }
16. }
17. // It is not good.
18. public class Serializable: Attribute
19. {
20. ….
21. }

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do   **Name types using nouns, noun phrases or adjective phrases.**

**Why:**

Bad examples include SearchExamination (a page for searching for examinations), Common (does not end with a noun, and does not explain its purpose) and SiteSecurity (although the name is technically okay, it does not say anything about its purpose).

Good examples include BusinessBinder, SmartTextBox, or EditableSingleCustomer.

Do not   **repeat the name of a class or enumeration in its members.**

1. public class Employee
2. {
3. // Wrong!
4. public static Employee GetEmployee()
5. {
6. ….
7. }
8. public bool DeleteEmployee()
9. {
10. ….
11. }
12. // Right
13. public static Employee Get()
14. {
15. ....
16. }
17. public bool Delete()
18. {
19. ....
20. }
21. public void AddNewJob()
22. {
23. ....
24. }
25. public void RegisterForMeeting()
26. {
27. ....
28. }
29. }

**Why:** it is concisely and easy to use.

Do not   **include** **numbers in** **variables, parameters and type members.**

1. //it is not good
2. public string status1;
3. public void GetStatus(string key1,string key2)
4. {
5. ….
6. }

**Why:** it is better to use a meaningful name than number variables.

Do not  **short names or names that can be mistaken with other names.**

1. //it is not good
2. public string variable;
3. public string StringBuilder;

**Why:** It is not important if the name is short or not, we must make sure the name is meaningful, because it is no comments in myte application.

Do not  **use abbreviation to name variables, parameters and type members.**

1. //it is not good
2. public string EAS;
3. //it is good
4. public string ExpenseApprovalSatus;

**Why:** It is understandable for developer who doesn’t know the abbreviation.

## Member Design Guidelines

Do  **Allow properties to be set in any order.**

**Why:** the order is not important for application and developers, so we don’t care about that.

Do  **Use a** **method instead of a property.**

* If the work is more expensive than setting a field value.
* If it represents a conversion such as the Object.ToString method.
* If it returns a different result each time it is called, even if the arguments didn’t change. For example, the NewGuid method returns a different value each time it is called.
* If the operation causes a side effect such as changing some internal state not directly related the property (which violates the Command Query Separation).

Populating an internal cache or implementing lazy-loading is a good exception.

**Why:** Method can restrict the functionality, implement complex business and easy to expand, however property can’t do that.

Do  **A method or property should do only one thing.**

**Why:** It is easy to understand, and it is easy extend and reusing.

Do  **Return an IEnumerable<T> or ICollection<T> instead of a concrete collection class.**

**Why:**

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>.

If you’re using .NET 4.5, you can also use IReadOnlyCollection<T>, IReadOnlyList<T> or IReadOnlyDictionary<TKey, TValue>.

Do  **Properties, methods and arguments representing strings or collections should never be null.**

**Why:**

Returning null can be unexpected by the caller. Always return an empty collection or an empty string instead of a null reference. This also prevents cluttering your code base with additional checks for null, or even worse, string.IsNotNullOrEmpty().

Do  **Define parameters as specific as possible.**

**Why:**

If your member needs a specific piece of data, define parameters as specific as that and don’t take a container object instead. For instance, consider a method that needs a connection string that is exposed through some central IConfiguration interface. Rather than taking a dependency on the entire configuration, just define a parameter for the connection string. This not only prevents unnecessary coupling, it also improved maintainability in the long run.

Do not   **use #region statements.**

**Why:** It’s easy to distinguish and search different models. We can use function instead of #region in this case.

Do not   **use mutual exclusive properties.**

See below code:

1. public class People
2. {
3. // Below two methods are not good.
4. public bool CanSpeakEnglish()
5. {
6. ….
7. }
8. public bool CanNotSpeakEnglish()
9. {
10. ….
11. }
12. }

**Why:** It is repetitive and easy make developer confusing.

## Miscellaneous Design & Layout Conventions

Do  **All class field should be declared at the top, with one line separating them from the properties or methods.**

1. public class MyClass
2. {
3. // At top of the class
4. private int \_number;
5. private string \_name;
6. public void SomeMethod( )
7. {
8. ….
9. }
10. public void SomeMethodOne( )
11. {
12. ….
13. }
14. }

**Why:** because the fields can be used in whole class, the developers can easy to find the fields which they needed. It is easy to miss when the fields are not at the top of class.

Do  **Specifically add Access Modifiers in front of private field/property/method/class/interface/enum declaration.**

1. public class MyClass
2. {
3. private int \_number;
4. …
5. }

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do  **Always place an open curly brace (‘{‘, ‘}’) in a new line.**

1. // Bad sample
2. public void Clear(){\_validationRules.Clear();}
4. // Good sample
5. public void Clear()
6. {
7. \_validationRules.Clear();
8. }

**Why:** It is easy to read.

Do  **Initialize string to** **String.Empty rather than assigning it "".**

1. // It is good.
2. public string UserName = String.Empty;
3. // It is not good.
4. public string UserName = "";

**Why:** This two can be used, but String.Empty is more readability.

Do   **Avoid using the method ".Equals()" to compare 2 strings, use "==" instead.**

1. var myString1 = null;
2. var myString2 = String.Empty;
4. // It is good.
5. if (myString1 == myString2)
6. // It is not good.
7. if (myString1.Equals(myString2)) --> posible Null Reference Exception!

**Why:** For myString1.Equals(myString2), if myString1 is null, the sentence will raise a Null Reference Exception.

Do   **Dictionary usage - Avoid using ".Add()" method to add new values inside a key.**

Example:

1. var myDictionary = new Dictionary<string, List<MyValue>>();
3. var value1 = new MyValue{ Value = "1"};
4. var value2 = new MyValue{ Value = "2"};
5. var value3 = new MyValue{ Value = "3"};
7. var values = new List<MyValue>();
9. values.Add(value1);
10. values.Add(value2);
11. values.Add(value3);
13. Good:
14. myDictionary["myKey"] = myValues;
16. Not good:
18. if (!myDictionary.ContainsKey["myKey"])
19. {
20. myDictionary.Add("myKey", myValues )   --> Possible concurrency error: "An item with the same key has already been added."
21. }
22. else
23. {
24. myDictionary["myKey"] = myValues;
25. }

**Why:** because it can lead to concurrency errors. Use direct assignation instead, as long as we are sure that the values are constant.

Do  **Dictionary usage - When referencing a specific key, make sure it exists.**

Example:

1. var myDictionary = new Dictionary<string, List<MyValue>>();
3. var value1 = new MyValue{ Value = "1"};
4. var value2 = new MyValue{ Value = "2"};
5. var value3 = new MyValue{ Value = "3"};
7. var values = new List<MyValue>();
9. values.Add(value1);
10. values.Add(value2);
11. values.Add(value3);
13. myDictionary["myKey"] = myValues;
15. Not good:
17. var myValues = myDictionary["KeyThatDoesNotExist"]; --> Error! The given key was not present in the Dictionary.
19. Good:
21. List<MyValue>() myValues;
23. if (myDictionary.ContainsKey("KeyThatDoesNotExist"))
24. {
25. myValues = myDictionary["KeyThatDoesNotExist"];
26. }

**Why:** It can keep the application robustness.

Do  **Always useTryParse() instead of ConvertTo().**

Example:

1. var myString = String.Empty;
2. DateTime myStringConvertedToDateTime;
4. Good:
6. if (DateTime.TryParse(myString, out myStringConvertedToDateTime) == false)  
   {  
    myStringConvertedToDateTime=DateTime.MinValue; //Add a default value or handled exception here.  
   }
8. Not good:
9. myStringConvertedToDateTime = Convert.ToDateTime(myString);

**Why:** For ConvertTo, if the parameter is null or empty, it will throw an exception by system, but we can handle and prevent the exception by using TryParse.

Do  **Using WHERE for** **Linq/Lambda Expressions.**

|  |  |
| --- | --- |
| Incorrect Case | Correct Case |
| myCollection.Where(person => person.Age > 25).First(); | myCollection.First(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).FirstOrDefault(); | myCollection.FirstOrDefault(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).Any(); | myCollection.Any(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).Single(); | myCollection.Single(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).SingleOrDefault(); | myCollection.SingleOrDefault(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).Count(); | myCollection.Count(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).Last(); | myCollection.Last(person => person.Age > 25); |
| myCollection.Where(person => person.Age > 25).LastOrDefault(); | myCollection.LastOrDefault(person => person.Age > 25); |

**Why:** WHERE operator is used to filter elements within a collection. This operator goes through the entire collection, evaluating the conditional clause that we programmed for each element.  
To avoid performance issues, when using WHERE operator we must ask ourselves if it is really necessary to go through the entire collection to then be able to perform an action over the filtered elements or if we can simply use another operator that doesn’t go through the entire collection, and in this way execute only one action (the new operator) instead of two (WHERE + the new operator).

Do  **Using** ANY **for** **Linq/Lambda Expressions.**

|  |  |
| --- | --- |
| InCorrect Case | Correct Case |
| if (myCollection.Count() >​ 0) | if (myCollection.Any()) |
| if (myCollection.Count() == 0) | if (!myCollection.Any()) |

**Why:** ANY operator is used to determine if at least one object in a collection fulfills a specific condition. Unlike WHERE and .Count() methods, ANY operator doesn’t go through the entire collection, it stops once it finds an element with the required condition. It is always more performing to use ANY rather than .Count().

Do  **Write only one statement per line. Write only one declaration per line.**

1. // It is good.
2. public string UserName;
3. public string UserAddress;
4. // It is not good.
5. public string UserName,UserAddress;

**Why:** It is not readable and not in line with .net framework coding standards.

Do  **Tabs are used for indentation not spaces.**

**Why:** It’s easy to read and maintain the code.

Do  N**ever hardcode a numeric value; always declare a constant instead.**

1. // It is good.
2. var BuildVersion = 1;
3. BuildTheTrunk(BuildVersion);
4. // It is not good.
5. BuildTheTrunk(1);

**Why:** define a constant for a numeric value can make it meaningful and easy to change for developer.

Do  **Use a try-catch statement for most exception handling.** **Catch only exceptions for which you have explicit handling.**

1. // It is good.
2. try
3. {
4. result = GetInstance(code);
5. }
6. catch (Exception)
7. {
8. throw new Exception(string.Format("Cannot cast code: &apos;{0}&apos;", code));
9. }
10. // It is not good.
11. try
12. {
13. result = GetInstance(code);
14. }
15. catch (Exception)
16. {
18. }

**Why:** If we use the try-catch statement, we must handle the exception manually, otherwise we may be lose some errors.

Do  **Simplify your code by using the C# using statement. If you have a try-finally statement in which the only code in the finally block is a call to the Dispose method, use a using statement instead.**

1. Font font1 = new Font("Arial", 10.0f);
2. try
3. {
4. byte charset = font1.GdiCharSet;
5. }
6. finally
7. {
8. if (font1 != null)
9. {
10. ((IDisposable)font1).Dispose();
11. }
12. }
13. // You can do the same thing with a using statement.
14. using (Font font2 = new Font("Arial", 10.0f))
15. {
16. byte charset = font2.GdiCharSet;
17. }

**Why:** using statement can recycle the memory immediately automatically.

Do  **It is optional to** **use a curly brace scope in if/while/for statement.**

1. // It is good.
2. for (int i = 0 ; i<10 ; i++)
3. {
4. ...
5. }
6. if(condition)
7. {
8. ...
9. }
10. // It is ok
11. for (int i = 0 ; i<10 ; i++)
12. count ++;
13. // It is not good.
14. for (int i = 0 ; i<10 ; i++)
15. count ++;
16. // It is not good.
17. for (int i = 0 ; i<10 ; i++) count ++;
18. // It is not good.
19. for (int i = 0 ; i<10 ; i++) {...}
20. if(condition){...}

**Why:** not all cases use the curly brace scope, for example only one sentence, but we must keep the sentence in a single line and leave 4 blanks space before the sentence.

Do  **Avoid boxing and unboxing value types.**

1. // It is not good.
2. //boxing
3. object number = 10;
4. object week = Week.Monday;
5. //unboxing
6. int index = (int)number;
7. // It is good.
8. int number = 10;
9. Week week = Week.Monday;

**Why:** boxing and unboxing can cause the performance issue.

Do  **Avoid using the trinary conditional operator.**

1. // It is not good.
2. private bool ShouldShowPpaReason(TimeReportStatus status)
3. {
4. return status == TimeReportStatus.DraftAdjustment &&
5. ((BaseTimeReportPage.LoggedInUser.SettingConstraintForCurrentPeriod != null ?
6. BaseTimeReportPage.LoggedInUser.SettingConstraintForCurrentPeriod.Company == "1010"
7. : BaseTimeReportPage.LoggedInUser.ActiveCompanyCode == "1010") ||
8. \_HasFedProject());
9. }
10. // It is ok.
11. bool tag = count > 0 ? true : false;

**Why:** it is hard to read for a complex trinary conditional operator, but we can still use the easy trinary operator.

Do  **Use the “****@” prefix for string literals instead of escaped strings.**

1. // It is good.
2. public const string EXPENSE\_SHOULD\_BE\_CHARGED\_THROUGH\_AMEX = @"This expense\ex07 should be charged through AMEX";
3. // It is not good.
4. public const string EXPENSE\_SHOULD\_BE\_CHARGED\_THROUGH\_AMEX = "This expense\ex07 should be charged through AMEX";

**Why:** @ will ignore the escape character.

Do  **When building a long string, use** **StringBuilder, not string.**

1. var sqlStringBuilderQuery = new StringBuilder();
2. sqlStringBuilderQuery.Append("SELECT jc FROM JobCode jc WHERE jc.Number = ? ");
3. sqlStringBuilderQuery.Append(afterCareerDate ?  "and jc.CareerLevelGroupId =?": "and jc.WorkForceCd =? ");

**Why:** StringBuilder have higher performance than string, especially in a [loop](app:ds:loop) [statement](app:ds:statement).

Do  **Never concatenate strings inside a loop. Remember, strings are immutable. Each time you concatenate, a new instance of string is created.**

See below code:

1. // It is not good.
2. string strCustomerName = "";
3. for (var i = 0 ; i < Customer.Count ; i++)
4. {
5. strCustomerName += Customer[i].Name;
6. }
7. // It is good.
8. StringBuilder strCustomerName = new StringBuilder();
9. for (var i = 0 ; i < Customers.Count ; i++)
10. {
11. strCustomerName.Append(Customers[i].Name);
12. }

**Why:** Cause strings are immutable, if you create a new string then you can’t modify it and you have to create a new object that will allocate some new space and take up system, so we use string builder and also use append connection string. Because string builder can modify without create a new object. (Before reaching the ceiling will not assign the new space) default is 16 “insert “, “remove”,” replace”.

Do  **Never hardcode strings that will be presented to end users.**

See below code:

1. public class ProjectApprovalError
2. {
3. string errorMessage = @"Approvee already exists.";
4. public string GetErrorMessage(bool singleOrMultiple)
5. {
6. if (ImpactUserEidList.Count > 0)
7. {
8. switch (Type)
9. {
10. StringBuilder strNewCustomerName = new StringBuilder(“0123”,20)
11. for (var j = 0 ; j < newCustomer.Count ; i++)
12. {
13. strCustomerName.Append(newCustomer[j]);
14. }
15. for (var i = 0 ; i < Customers.Count ; i++)
16. {
17. strCustomerName.Append(Customers[i].Name);
18. }
19. }
20. }
21. }
22. }

**Why** **:** if you use hardcode strings it is hard for maintain, because if you want change that you must locate to the place where you defined it , but if you use a variable, you can just modify this var then all the position where used this var will follow this operation.

Do  **Use the + operator to concatenate short strings.**

**Why:** if you defined a variable is not string, use + operator can changed it to string auto.

Simple:

var h = Aaron;

string Demo = “hello” + h +”nice”;

Do  **Prefer to use the** as **operator and check for null, rather than directly casting, and having to handle potential** **[InvalidCastException](http://msdn.microsoft.com/en-us/library/system.invalidcastexception.aspx).**

1. // The good one
2. TimeCategoryValidationSource valideSource = errorSource as TimeCategoryValidationSource;
3. If(validSource != null)
4. {
5. …
6. }
7. // The bad one
8. object valideSource = new TimeCategoryValidationSource ();
9. TimeCategoryValidationSource  newValue = null;
10. try
11. {
12. newValue = (TimeCategoryValidationSource) valideSource;
13. }
14. catch( Exception e )
15. {
16. MessageBox.Show( e.Message );
17. }

**Why:** when we use as operator, if object is null, then it will not throw an exception, Instead it is to return null.

Do  **If a class or a method offers both generic and non-generic flavors, always prefer using the generics flavor.**

1. // It is good.
2. var charges = new List<ExpenseChargeForProjectDtoForSave>();
3. // It is not good.
4. var charges = new List();

**Why:** Do not need to determine one or more specific parameters, specific parameters which may delay statement, client code to achieve.

Do   **Make only the most necessary types public. Mark other as internal or private.**

**Why:** Object-oriented encapsulation,use private is useful for the security example :it can protect property in the class , other user can’t modify those property.

Do   **Provide a rich and meaningful exception message text.**

1. private Country \_FindCountryByCountryKey(string countryKey)
2. { var countryList = \_GetHibernateTemplate().Find<Country>(@"FROM country C WHERE C.Key =?",countrykey);
3. if (countryList.Count == 0)
4. {
5. throw new OgteObjectNotFoundException(@"Country not found for country key" + countryKey);
6. }
7. return countryList[0];
8. }

**Why:** It is easy to find the error code.

Do   **Throw the most specific exception that is appropriate.**

See below code:

1. DateTime dateDisplayParsed;
2. if (!DateTime.TryParse(amexDto.DateDisplay, out dateDisplayParsed))
3. {
4. // It is good.
5. throw new FormatException(String.Format("Could not parse date {0} ", amexDto.DateDisplay));
6. // It is not good.
7. throw new Exception(String.Format("Could not parse date {0} ", amexDto.DateDisplay));
8. }

**Why:** It is for developer to locate the error code and find the reason.

Do   **Use using statements instead of fully qualified type names.**

**Why:** Limit usage of fully qualified type names to prevent name clashing. For example, don’t do this

1. var list = new System.Collections.Generic.List<string>();

Instead, do this

1. using System.Collections.Generic;
2. var list = new List<string>();

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

1. using Label = System.Web.UI.WebControls.Label;

Do   **Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.**

**Why:** Only use var as the result of a LINQ query, or if the type is very obvious from the same statement and using it would improve readability.

**So don't**

1. var i = 3; // what type? int? uint? float?
2. var myfoo = MyFactoryMethod.Create("arg");
3. // Not obvious what base-class or
4. // interface to expect. Also difficult
5. // to refactor if you can't search for
6. // the class

Instead, use var like this.

1. var q = from order in orders where order.Items > 10 and order.TotalValue > 1000;
2. var repository = new RepositoryFactory.Get<IOrderRepository>();
3. var list = new ReadOnlyCollection<string>();

Do   **Use implicit typing to determine the type of the loop variable in for and foreach loops.**

1. var syllable = "ha";
2. var laugh = string.Empty;
3. for (var i = 0; i < 10; i++)
4. {
5. laugh += syllable;
6. Console.WriteLine(laugh);
7. }

The following example uses implicit typing in a foreach statement.

1. foreach (var ch in laugh)
2. {
3. if (ch == 'h')
4. {
5. Console.Write("H");
6. }
7. else
8. {
9. Console.Write(ch);
10. }
11. }
12. Console.WriteLine();

**Why:** The following example uses implicit typing in a “for” statement.

Do  **Declare and initialize variables as late as possible.**

See below code:

1. public TimesheetInfo GetTimesheetByLoggedInUser(ILoggedInUser loggedInUser)
2. {
3. // It is not good define timesheetInfo at beginning of the function.
4. var timesheetInfo = new TimesheetInfo();
5. ....
6. ....
7. ....
8. // It is good define timesheetInfo here instead of defining at beginning of the function.
9. var timesheetInfo = new TimesheetInfo();
10. GetWorkScheduleOrShiftScheduleUrl(timesheetInfo, loggedInUser);
11. AddTimeCategoryTasksAndMetadataToTimesheetInfo(timesheetInfo, timesheetDtoForDisplayBuilder);
12. }

**Why:** no.

Do   **Assign each variable in a separate statement.**

Don’t use confusing constructs like the one below. Below code is not good:

1. var result = someField = GetSomeMethod();

**Why:** It is hard to read and easy to miss some variables.

Do   **Favor Object and Collection Initializers over separate statements.**

Instead of

1. var startInfo = new ProcessStartInfo(“myapp.exe”);
2. startInfo.StandardOutput = Console.Output;
3. startInfo.UseShellExecute = true;

Use Object Initializers

1. var startInfo = new ProcessStartInfo(“myapp.exe”)
2. {
3. StandardOutput = Console.Output,
4. UseShellExecute = true
5. };

Similarly, instead of

1. var countries = new List<string>();
2. countries.Add(“Netherlands”);
3. countries.Add(“United States”);

Use collection or dictionary initializers

1. var countries = new List<string>
2. {
3. “Netherlands”,
4. “United States”
5. };

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do  **Always add a default block** **after the last case in a switch statement.**

1. void Foo(string answer)
2. {
3. switch (answer)
4. {
5. case "no":
6. Console.WriteLine("You answered with No");
7. break;
8. case "yes":
9. Console.WriteLine("You answered with Yes");
10. break;
11. default:
12. // Not supposed to end up here.
13. throw new InvalidOperationException("Unexpected answer " + answer);
14. }
15. }

**Why:** Add a descriptive comment if the default block is supposed to be empty. Moreover, if that block is not supposed to be reached throw an InvalidOperationException to detect future changes that may fall through the existing cases. This ensures better code, because all paths the code can travel has been thought about.

Do  **Be** **reluctant with multiple return statements.**

It is not good for below code:

1. private static int FindScheduleSettingIndexForWorkSchedule(IList<ScheduleSetting> settings, WorkSchedule workSchedule, bool evaluateByType)
2. {
3. for (var i = 0; i < settings.Count; i++)
4. {
5. if (evaluateByType)
6. {
7. if (settings[i].WorkSchedule.GetType().Equals(workSchedule.GetType()))
8. {
9. return i;
10. }
11. }
12. else if (settings[i].WorkSchedule.Id == workSchedule.Id)
13. {
14. return i;
15. }
16. }
17. return -1;
18. }

**Why:** It will break the code structure, hard to maintain the code and easy to miss calling some method.

Do   **Encapsulate complex expressions in a method or property.**

**Why:** Consider the following example:

1. //It is not good.
2. if (member.HidesBaseClassMember && (member.NodeType != NodeType.InstanceInitializer))
3. {
4. // do something
5. }

In order to understand what this expression is about, you need to analyze its exact details and all the possible outcomes. Obviously, you could add an explanatory comment on top of it, but it is much better to replace this complex expression with a clearly named method:

1. // It is good.
2. if (NonConstructorMemberUsesNewKeyword(member)) {
3. // do something
4. }
5. private bool NonConstructorMemberUsesNewKeyword(Member member) {
6. return (member.HidesBaseClassMember && (member.NodeType != NodeType.InstanceInitializer);
7. }

You still need to understand the expression if you are modifying it, but the calling code is now much easier to grasp.

Do  **Security - Every page (aspx, aspx.cs) must be created within OneGlobal.Web\secure.**

**Why:** According to the existing pages.

Do  **Security - When creating a page we must verify if it can be accessed exclusively by a group of users (Admin, SuperUsers, etc). If we need to validate access, we must do so within Page\_Load() method of aspx.cs.**

Example:

1. private void \_ValidateAccessPage()
2. {
3. if (!user.IsActualUserASuperUser())
4. {
5. Response.Redirect(MyteGlobalContants.TimesheetPage, true);
6. }
7. }
9. Correct Case (validate acces in each request):
11. protected void Page\_Load(object sender, EventArgs e)
12. {
13. \_ValidateAccessPage();
14. }
16. Incorrect Case (validate Access only the first time):
18. protected void Page\_Load(object sender, EventArgs e)
19. {
20. if (!Page.IsPostback())
21. {
22. \_ValidateAccessPage();
23. }
24. }

**Why:** It is the common way for myte to handle the user access, the first case can cover all visit.

Do not   **compare to true or false use if (condition).**

1. // It is good.
2. if (!HasRecipients())
3. {
4. if (requireEmailAddressees)
5. {
6. throw new InvalidOperationException("You may not send an email that has no recipients.");
7. }
8. }
9. // It is not good.
10. if (HasRecipients() == false)
11. {
12. if (requireEmailAddressees == true)
13. {
14. throw new InvalidOperationException("You may not send an email that has no recipients.");
15. }
16. }

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do not   **files with more than 200 lines (excluding machine-generated code). Methods with more than 25 lines.** **Lines exceed 80 characters.**

**Why:** no.

Do not   **cyclic references between assemblies.**

**Why:** A good code frame will not cause the cyclic references. The cyclic references may cause build issue.

Do not   **Methods exceed 7 statements.**

**Why:** A method that requires more than 7 statements is simply doing too much or has too many responsibilities. It also requires the human mind to analyze the exact statements to understand what the code is doing. Break it down in multiple small and focused methods with self-explaining names, but make sure the high-level algorithm is still clear.

Do not   **conditions with double negatives.**

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

1. bool hasOrders = !customer.HasNoOrders;

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

Do not   **use "****magic” numbers.**

**Why:** Don’t use literal values, either numeric or strings, in your code other than to define symbolic constants. For example:

1. public class Whatever {
2. public static readonly Color PapayaWhip = new Color(0xFFEFD5);
3. public const int MaxNumberOfWheels = 18;
4. }

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 example:

1. mean = (a + b) / 2; // okay
2. 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.

1. public class SomeSpecialContainer
2. {
3. public const int MaxItems = 32;
4. public const int HighWaterMark = 3 \* MaxItems / 4; // at 75%
5. }

**Note**An enumeration can often be used for certain types of symbolic constants.

Do not   **rely on the variable name to specify the type of the variable. It might not be correct.**

1. // Naming the following variable inputInt is misleading.
2. // It is a string.
3. var inputInt = Console.ReadLine();
4. Console.WriteLine (inputInt);

**Why:** It is redundant and we can easily know the variable type in higher version Visual Studio.

Do not   **change a loop variable inside a ‘for’ or ‘foreach’ loop.**

1. for (int index = 0; index < 10; ++index)
2. {
3. if (some condition)
4. {
5. index = 11; // Wrong! Use ‘break’ or ‘continue’ instead.
6. }
7. }

**Why:** 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. Although this rule also applies to foreach loops, an enumerator will typically detect changes to the collection the foreach loop is iteration over.

Do not  **nested loops.**

1. // Bad sample
2. foreach (var i in list1)
3. {
4. foreach (var j in list2)
5. {
6. DoSomething(i, j);
7. }
8. }
10. // Good sample
11. list1.ForEach(a => list2.ForEach(b => DoSomething(a, b)));

**Why:** A method that nests loops is more difficult to understand than one with only a single loop. In fact, in most cases having nested loops can be replaced with a much simpler LINQ query that uses the “from” keyword twice or more to join the data.

Do not  **use if-else statements instead of a simple (conditional) assignment.**

Express your intentions directly. For example, rather than bool pos;

1. if (val > 0)
2. {
3. pos = true;
4. }
5. else
6. {
7. pos = false;
8. }

Write

1. bool pos = (val > 0); // initialization

Or

1. if (val > 0)
2. {
3. pos = true;
4. }
5. else
6. {
7. pos = false;
8. }

Write

1. bool pos = (val > 0); // initialization

Or instead of

1. string result;
2. if (someString != null)
3. {
4. result = someString;
5. }
6. else
7. {
8. result = “Unavailable”;
9. }
10. return result;

Write

1. return someString ?? “Unavailable”;

**Why:** It is easy to read and keep the code concise.

Do not  **allow methods and constructors with more than three parameters commonly, but it also depending on complexity methods.**

**Why:** The calling convention options determine the order in which arguments passed to functions are pushed onto the stack; which function, calling or called, removes the arguments from the stack; and the name-decorating convention that the compiler uses to identify individual functions.

Big method should be refactored. And another point, for methods which require more than 3 or 5 parameters, we could envelop them into a parameter class. Event handler should be a good example, EventArgs and all its inherited classes are wrapping all useful parameters for event handler method. Example as below:

Protected void Page\_Load(object sender, EventArgs e)

Do not **(Avoid) use ref or out parameters.**

**Why:** Passing types by reference (using out or ref) requires experience with pointers, understanding how value types and reference types differ, and handling methods that have multiple return values. Also, the difference between out and ref parameters is not widely understood. Although return values are commonplace and heavily used, the correct application of out and ref parameters requires intermediate design and coding skills. Library architects who design for a general audience should not expect users to master working with out or ref parameters.

## Object Design Guidelines

Do  **A class or interface should have a single purpose, a file name should reflect the class/interface it contains.**

See below example:

1. interface ICloneable
2. {
3. object Clone( );
4. object DeepClone();
5. }
6. interface IComparable
7. {
8. int CompareTo(object other);
9. }
10. class ListEntry: ICloneable, IComparable
11. {
12. object ICloneable.Clone( )
13. {
14. …
15. }
16. object DeepClone()
17. {
18. …
19. }
20. int IComparable.CompareTo(object other)
21. {
22. …
23. }
24. }

Interface ICloneable only do one thing clone object, and Interface IComparable do another thing compare two objects.

**Why:** Interfaces should have a name that clearly explains the purpose or role of that interface within the system. Do not combine many vaguely related members on the same interface just because they were all on the same class. Separate the members based on the responsibility of those members so that callers only need to call or implement the interface related to a particular task.

Do  **Group all framework namespaces together and put custom or third-party namespaces under.**

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Web.UI.WebControls;
5. using Infragistics.Web.UI.GridControls;
6. using OneGlobal.Controller.Secure;
7. using OneGlobal.Model.Domain;
8. using OneGlobal.Model.Dto;
9. using OneGlobal.Web.Utils;
10. using OneGlobal.Web.Utils.Holidays;

We put .net framework namespace together and at the top.

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do  **The practical limit is probably 12. Do not have more than 20 members per interface.**

**Why:** If is better to read and use, it is means the interface need be refactor if it have more than 20 members.

Do  **Use an interface to decouple classes from each other.**

The interface is main approach to decouple classes and we use the Spring.Net to implement Dependency Injection in Myte solution.

**Why:** Interfaces are a very effective mechanism for decoupling classes from each other.

* They can prevent bidirectional associations;
* They simplify the replacement of one implementation with another;
* They allow replacing an expensive external service or resource with a temporary stub for use in a non-production environment.
* They allow replacing the actual implementation with a dummy implementation or a fake object in a unit test;
* Using a dependency injection framework you can centralize the choice which class is going to be used whenever a specific interface is requested.

Do  **It should be possible to** **treat a derived object as if it were a base class object.**

See below code:

1. public class People
2. {
3. public void Eat()
4. {
5. ….
6. }
7. }
8. public class ChinaPeople: People
9. {
10. ....
11. }
12. public class USAPeople: People
13. {
14. ....
15. }
16. public void PeerDevelop(People chinaPeople, People usaPeople)
17. {
18. ....
19. }

**Why:**

In other words, you should be able to use a reference to an object of a derived class wherever a reference to its base class object is used without knowing the specific derived class. A very notorious example of a violation of this rule is throwing a NotImplementedException when overriding some of the base-class methods. A less subtle example is not honoring the behavior expected by the base-class.

Do not   **putting multiple classes** **in a single file.**

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do not   **exposing the other objects an object depends on.**

1. someObject.SomeProperty.GetChild ().Foo ();

**Why:**

An object should not expose any other classes it depends on because callers may misuse that exposed property or method to access the object behind it. By doing so, you allow calling code to become coupled to the class you are using, and thereby limiting the chance you can easily replace it in a future stage. Using a class that is designed using the Fluent Interface pattern does seem to violate this rule, but it is simply returning itself so that method chaining is allowed. Inversion of Control or Dependency Injection frameworks often require you to expose a dependency as a public property. As long as this property is not used for anything else than dependency injection I would not consider it a violation.

Do not   **bidirectional dependencies, If we must use it according to our requirements, we need make sure it is very limited.**

Below two classes use the bidirectional dependencies.

1. public class Order
2. {
3. private Customer \_customer;
4. public Customer GetCustomer()
5. {
6. return \_customer;
7. }
8. public void SetCustomer(Customer arg)
9. {
10. if(\_customer != null)
11. {
12. \_customer.friendOrders().remove(this);
13. }
14. \_customer = arg;
15. if(\_customer != null)
16. {
17. \_customer.friendOrders().add(this);
18. }
19. }
20. }
21. public class Customer
22. {
23. private ISet \_orders = new HashSet();
24. public void addOrder(Order arg)
25. {
26. arg.setCustomer(this);
27. }
29. public ISet FriendOrders()
30. {
31. return \_orders;
32. }
33. }

**Why:**

This means that two classes know about each other’s public members or rely on each other’s internal behavior. Refactoring or replacing one of those two classes requires changes on both parties and may involve a lot of unexpected work. The most obvious way of breaking that dependency is introducing an interface for one of the classes and using Dependency Injection.

Sometimes it is really hard to avoid bidirectional dependencies, we believe the use should be very limited, and very few unavoidable use should be known to all. For example Expense and TimeReport relationship in myte it is a [particular](app:ds:particular) [case](app:ds:case).

Do not   **putting a using statement inside a namespace.**

It is good put the using statement outside the namespace.

1. using System;
2. using System.Collections.Generic;
3. using System.Net.Mail;
4. namespace OneGlobal.Model.Common
5. {
6. ….
7. }

**Why:** Consistent with the C# Coding Standards of Microsoft's .NET Framework.

Do not   **hide** **inherited members with the new keyword.**

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:

1. public class Book
2. {
3. public virtual void Print()
4. {
5. Console.WriteLine("Printing Book");
6. }
7. }
8. public class PocketBook : Book
9. {
10. public new void Print()
11. {
12. Console.WriteLine("Printing PocketBook");
13. }
14. }

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

1. PocketBook pocketBook = new PocketBook();
2. pocketBook.Print(); // Will output "Printing PocketBook "
3. ((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.

**Why:**

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the virtual modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the override keyword or hide the virtual method in the base class by using the new keyword. If the method in the derived class is preceded with the new keyword, the method is defined as being independent of the method in the base class.