C # CODING STANDARDS & GUIDELINES

BY

ADEEL FAKHAR – DEVELOPMENT MANAGER

Why Coding Standards?

Coding standards serve the following purposes:

- They create a consistent look to the code, so that readers can focus on content, not layout.
- They enable readers to understand the code more quickly by making assumptions based on previous experience.
- They facilitate copying, changing, and maintaining the code.
- They demonstrate C# best practices.

There are some golden rules to achieve good coding standards.

- Naming Conventions
- Layout/Formatting Conventions
- Commenting Conventions
- Optimizing Syntax

Let’s discuss them, one by one.

Check	Severity	1st Level	2nd Level
Naming Conventions	Medium	✓	✓

Naming is hard. And because it's hard, I don't want to confound the issue with mixed casing. Every language has its own casing standards. To add to the confusion, many organizations, or even teams, have their own standards. Follow the conventions that are being widely used & strict to them.

There are three casings generally accepted by C# Standards:

- PascalCase
- camelCase
- UPPER_CASE

1. **Class and Method names should always be in Pascal Case**

```
public class Employee
{
    public Employee GetDetails()
    {
        //...
    }
    public double GetBonus()
    {
        //...
    }
}
```

2. **Method argument and Local variables should always be in Camel Case**

```
public class Employee
{
    public void PrintDetails(int employeeId, String firstName)
    {
        int totalSalary = 2000;
        // ...
    }
}
```

3. **Avoid the use of underscore while naming identifiers**

```
// Correct
```

```
public DateTime fromDate;
public String firstName;

// Avoid
public DateTime from_Date;
public String first_Name;
```

4. **Always prefix an interface with letter I.**

```
// Correct
public interface IEmployee
{
}
public interface IShape
{
}
public interface IAnimal
{
}

// Avoid
public interface Employee
{
}
public interface Shape
{
}
public interface Animal
{
}
```

5. **Constants should always be declared in UPPER_CASE.**

```
// Correct
public const int MIN_AGE = 18;
public const int MAX_AGE = 60;

// Avoid
```

```
public const int Min_Age = 18;
public const int Max_Age = 60;
```

6. **Do not use Hungarian notation or any other type identification in identifiers (except interfaces):**

```
int iCounter;
string strName;
string spCreateUsers;
OrderingService svcOrdering;
```

Hungarian notation invites two problems in C#. For one, the name is misleading when the type changes. Another problem is readability. Visual Studio code editor already provides helpful tooltips to determine object types. In general, you want to avoid type indicators in the identifier.

7. **Do use meaningful and self-explanatory names for classes, methods and properties:**

```
int daysUntilProgramExpiry;

public List<Person> GetPersonProfileById(long personId)
{
      //do something
}
```

This makes your code easier to read and understand without having you to write (or atleast minimizes) comments of what the code does.

8. **Do suffix asynchronous methods with the Async word:**

```
public async Task<List<Person>> GetPersonProfileByIdAsync(long personId)
{
    //do something
}
```

This enable developer to easily identify synchornous vs asynchronous methods by just looking at the method itself.

9. **Do prefix global variables and class members with underscores (_):**

```
private readonly ILogger<ClassName> _logger;
private long _rowsAffected;
```

```
private IEnumerable<Persons> _people;
```

This is to easily differentiate between local and global identifiers/variables.

10. **Try to use short-hand names only when they're generally known:**

```
private readonly CreateQuestionDefinitionRequestDto _requestDto;
```

It would be too much to name a variable "createQuestionDefinitionRequestDto" when you know that the variable/parameter is a request object. The same thing applies for FTP, UI, IO, etc. It's perfectly fine to use abbreviation for as long as they're generally known, otherwise it would be counter productive not to do so.

11. **Avoid underscores (_) in between identifier names:**

```
public PersonManager person_Manager;
private long rows_Affected;
private DateTime row_updated_date_time;
```

It's to be consistent with the Microsost .NET Framework convention and makes your code more natural to read. It can also avoid "underline stress" or inability to see underline.

12. **Do use singular form, noun or noun phrases to name a class:**

```
public class Person
{
   //some code
}

public class BusinessLocation
{
   //some code
}

public class DocumentCollection
{
   //some code
}
```

This enables you to easily determine if an object holds a single item value or collection. Imagine, if you have a List<People> vs List<Person>. It's just odd to put plural form names in a List or Collection.

13. Do use nouns or adjective phrases for Property names as well.

When naming boolean properties or variables, you may add the prefix "can", "is", "has", etc. just like in the following:

```
public bool IsActive { get; set; }
public bool CanDelete { get; set; }

//variables
bool hasActiveSessions = false;
bool doesItemExist = true;
```

Adding those suffixes will provide more value to the caller.

Check	Severity	1st Level	2nd Level
Layout/Formatting Conventions	High	✓	✓

We have formatting conventions for C#, just as we do in any programming language. Formatting has to do with line breaks, indentation, and spacing. True, the IDE will often help with formatting, but it doesn't do everything for you.

1. **Indentation**

Indentation is a hotly contested issue in many languages. Should you use tabs or spaces? This document will not settle the argument. But it will establish that four is the proper number of spaces. Also, if you're using tabs instead, the tab should be equal to four spaces. These are defaults in Visual Studio, and most IDEs have this preset for C#.

2. **Brackets**

In C#, brackets contain the scope of a namespace, class, interface, method, and the like. Brackets are major structural constructs and should always go on their own line. I am just trying to rationalize it, but honestly, it's just the common convention.

3. **Terminator**

A semicolon terminates a statement. It goes on the same line as the end of the statement, much like a period would. Don't use extra semicolons, one is enough.

4. **Extra Lines**

One extra line is sufficient for separating methods or different sections of a class/program such as the list of member variables, constructors, and methods.

```
// Correct
class Employee
{
private int employeeId { get; set; }

public void PrintDetails()
{
//------------
}
}
```

```
// Avoid
class Employee
{

private int employeeId { get; set; }



public void PrintDetails()
{
//------------
}

}
```

5. **Braces**

For better code indentation and readability always align the curly braces vertically.

```
// Correct
class Employee
{
   static void PrintDetails()
   {
   }
}

 // Avoid
class Employee
   {
   static void PrintDetails()
   {
     }
}
```

6. **General Conventions**

1. Use the default Code Editor settings & write only one statement per line.
2. Write only one declaration per line.
3. Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
if ((val1 > val2) && (val1 > val3))
{// Take appropriate action.}
```

Check	Severity	1st Level	2nd Level
Commenting Conventions	High	✓	✓

1. Place the comment on a separate line, not at the end of a line of code.

2. Begin comment text with an uppercase letter.

3. End comment text with a period.

4. Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

```
// The following declaration creates a query. It does not run
// the query.
```

5. Don't create formatted blocks of asterisks around comments.

6. Ensure all public members have the necessary XML comments providing appropriate descriptions about their behavior.

Check	Severity	1st Level	2nd Level
Optimizing Syntax	High		✓

1. **Avoid the traditional if-else statements like in the following:**

```
bool result;
if (condition)
{
  result = true;
}
else
{
  result = false;
}
```

Do use ternary conditional operator (?:) instead:

```
bool result = condition ? true: false;
```

The preceding code is much cleaner, easier to read and understand. On top of that, it's more concise.

2. **Avoid using if statement for null checks like in the following:**

```
if (something != null)
{
   if (other != null)
   {
      return whatever;
   }
}
```

Do use null conditional (?.) operator instead:

```
return something?.other?.whatever;
```

The preceding code is also much cleaner and concise.

3. **Try to avoid complex if-else statements for null checks like in the following:**

```
if (something != null)
{
   if (other != null)
   {
```

```
            return whatever;
        }
        else
        {
            return string.empty;
        }
    }
    else
    {
        return string.empty;
    }
```

Do use null coalescing (??) operator instead:

```
return something?.other?.whatever ?? string.empty;
```

4. Avoid **using the following code when returning a default value when an object is null:**

```
int? number = null;
var n = number.HasValue ? number : 0;
```

Do use null coalescing (??) operator as well:

```
var n = number ?? 0;
```

or alternatively, you could do:

```
var n = number.GetValueOrDefault();
```

5. Avoid **using the equality operator (==) or HasValue for nullable variable check like in the following:**

```
int? number = null;

if (number == null)
{
    //do something
}

if (!number.HasValue)
{
    //do something
}
```

While the preceding code is fine, we can still improve that by using the is keyword like in the following:

```
int? number = null;

if (number is null)
{
    //do something
}
```

The preceding code is much easier to read as the intent is very clear.

6. **Avoid code without braces ({}) for single conditional if statement, for and foreach loops like in the following:**

```
if(conditioin) action;
```

Without the braces, it is too easy to accidentally add a second line thinking it is included in the if, when it isn't.

Always use braces instead:

```
if (condition) { action; }

//or better
if (condition)
{
   action;
}
```

7. Avoid **using multiple if-else statements like in the following:**

```
if (condition)
{
  //do something
}
else if(condition)
{
  //do something
}
else if(condition)
{
  //do something
}
else(condition)
{
  //do something else
```

```
    }
```

Do use switch statements instead:

```
switch(condition)
{
  case 1:
    //do something
    break;
  case 2:
    //do something
    break;
  case 3:
    //do something
    break;
  default:
    //do something else
   break;
}
```

But prefer switch expressions over switch statements where possible like in the following:

```
condition switch
{
   1 => //do something;
   2 => //do something;
   3 => //do something;
   _ => //do something else;
}
```

The preceding code is more concise yet, still easy to read and understand. (Note, only available in C# 8 or later versions)

Exceptions - There are cases that if-else statements would make more sense than using switch statements. For example, if the condition involves different objects and complex conditions. Use your judgement to whatever works better.

8. Do use the "using" statement, it is your friend.

When you have a disposable type, there's a great little construct built into the language: "using". It calls Dispose when the program flow leaves the scope. Here's an example:

```
using(var connection = _connectionFactory.Get())
using(var stream = connection.OpenStream())
{
   // use the connection and the stream
}
```

This example takes a little explanation. For one thing, most people don't know you can stack the using statements like this. Another thing is when the program flow exits the brackets for any reason both the connection and the stream will be disposed of.

Hopefully, if you have a well-designed connection class, the connection will close prior to being disposed of. In this case, there's no need to do anything else. This is how C# is meant to be!

9. **Avoid concatenating strings with the + sign/symbol like in the following:**

```
string name = "Vianne";
string greetings = "Hello " + name + "!";
```

Use string.Format() method instead:

```
string name = "Vynn";
string greetings = string.Format("Hello {0}!", name);
```

Or prefer using string interpolation ($) instead where possible:

```
string name = "Vjor";
string greeting = $"Hello, {name}!;
```

The preceding code is much more concise and readable compared to other approaches.

10. Avoid **string.Format() when formatting simple objects like in the following:**

```
var date = DateTime.Now;
string greetings = string.Format("Today is {0}, the time is {1:HH:mm} now.", date.DayOfWeek,
date);
```

Use string interpolation instead:

```
var date = DateTime.Now;
string greetings = $"Today is {date.DayOfWeek}, the time is {date:HH:mm} now.");
```

The preceding code is much easier to understand and concise. However, there are certain cases that using the **string.Format()** would makes more sense. For example, when dealing with complex formatting and data manipulation. So, use your judgement when to apply them in situations.

11. **Avoid using specific type for complex objects when defining variables like in the following:**

```
List<Repository.DataAccessLayer.Whatever> listOfBlah = _repo.DataAccessLayer.GetWhatever();
```

Use the **var** keyword instead:

```
var listOfBlah = _repo.DataAccessLayer.GetWhatever();
```

Same goes for other local variables:

```
var students = new List<Students>();
var memoryStream = new MemoryStream();
var dateUntilProgramExpiry = DateTime.Now;
```

12. Avoid **one-liner method implementation with curly braces like in the following:**

```
public string Greeter(string name)
{
   return $"Hello {name}!";
}
```

Do use Expression-bodied (=>) implementation instead:

```
public string Greeter(string name) => $"Hello {name}!";
```

The preceding code is more concise while maintaining readability.

13. **Avoid object initialization like in the following:**

```
Person person = new Person();
person.FirstName = "Vianne";
person.LastName = "Durano";
```

Do use object and collection initializers instead:

```
var person = new Person {
        FirstName = "Vianne",
        LastName = "Durano"
};
```

The preceding code is more natural to read and the intent is clear because the properties are defined within braces.

14. **Avoid creating a class just to return two simple result sets like in the following:**

```
public Person GetName()
{
   var person = new Person
   {
      FirstName = "Vincent",
      LastName = "Durano"
   };

   return person;
}
```

Do use **Tuples** instead where possible:

```
public (string FirstName, string LastName) GetName()
{
   return ("Vincent", "Durano");
}
```

The preceding code is more convenient for accessing objects and manipulating the data set. Tuples replaces the need to create a new class whose sole purpose is to carry around data.

15. **Try to create an Extention Methods to perform common tasks such as conversion, validation, formatting, parsing, transformation, you name it. So, instead of doing the following:**

```
string dateString = "40/1001/2021";
var isDateValid = DateTime.TryParse(dateString, our var date);
```

The preceding code is perfectly fine and should handle the conversion safely. However, the code is bit lengthy just to do basic conversion. Imagine you have tons of the same code conversion cluttering within the different areas in your project. Your code could turn into a mess or potentially causes you a lot of development time overtime.

To prevent that, you should consider creating a helper/utility functions to do common tasks that can be reused across projects. For example, the preceding code can now be converted to following extension:

```
public static class DateExtensions
{
    public static DateTime ToDateTime(this string value)
       => DateTime.TryParse(value, out var result) ? result : default;
}
```

and you will be able to use the extension method like in the following anywhere in your code:

```
var date = "40/1001/2021".ToDateTime();
```

The preceding code makes your code concise, easy to understand and provides convenience

16. **Avoid using .NET predefined data types such as Int32, String, Boolean, etc.:**

```
String firstName;
Int32 orderCount;
Boolean isCompleted;
```

Do use built-in primitive data types instead:

```
string firstName;
int orderCount;
bool isCompleted;
```

Avoid the use of System data types and prefer using the Predefined data types. The preceding code is consistent with the Microsoft's .NET Framework and makes code more natural to read.

17. **Do not use initials as identifier abbreviations like in the following:**

```
private readonly PersonManager _pm;
```

The main reason for this is that it can cause confusion and inconsistency when you have class that might represents the same thing like in the following:

```
private readonly ProductManager _pm;
```

Instead, do choose clarity verbrevity like in the following:

```
private readonly PersonManager _personManager;
private readonly ProductManager _productManager;
```

The preceding code provides more clarity as it clearly suggests what the object is about.

18. **Do organize namespaces with a clearly defined structure.**

Generally, namespaces should reflect the folder hierarchy within a project. Take a look at the following example:

```
namespace ProjectName.App.Web
namespace ProjectName.Services.Common
namespace ProjectName.Services.Api.Payment
```

```
namespace ProjectName.Services.Api.Ordering
namespace ProjectName.Services.Worker.Ordering
```

The preceding code suggest good organization of your code within the project, allowing you to navigate between layers easily.

19. **Do declare all member variables and fields at the top of a class, with static fields at the very top:**

```
private static string _externalIdType;
private readonly ILogger<PersonManager> _logger;
private int _age;
```

This is just a generally accepted practice that prevents the need to hunt for variable declarations.

20. **Do consider putting all your private methods at the bottom after public methods:**

```
public class SomeClass
{
   private void SomePublicMethodA()
   {

   }

   // rest of your public methods here

   private void SomePrivateMethodA()
   {

   }

   private void SomePrivateMethodB()
   {

   }
}
```

Why? same reason that I mentioned for Tip # 19.

21. Avoid **grouping your code into regions like in the following:**

```
#region Private Members
   private void SomePrivateMethodA()
   {
```

```
    }

    private void SomePrivateMethodB()
    {

    }

#endregion
```

The preceding code is a code smell which could potentially make your code grow without you realizing it. I admit that I have used this feature many times to collapse the code within a class. However, I realize that hiding code into regions won't give you any value aside from maximizing your visual view when the region is collapsed. If you are working with a team of developers on a project, chances are, other developers will append their code in there until the code gets bigger and bigger over time. As a good practice, it's always recommended to keep your classes small as possible.

If you have tons of private methods within a class, you could split them into a separate class instead.

22. Try to use record types for immutable objects.

Record types is a new feature introduced in C# 9 where it simplifies your code. For example, the following code:

```
public class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

can be written in the following way using record:

```
public record Person(string FirstName, string LastName);
```

Using record types will automatically generate the boilerplate code for you and keeping your code concise. Records will be really useful for defining DTOs, Commands or any object that carries immutable data around

23. “var” Is Your Friend

Using?var is actually a good idea for all those short-lived variables. It not only saves a lot of redundancy, but it allows for more flexibility in the code. Basically, if you can already see the type based on the variable name, you’re good to go with var! Here are a few examples of var in practice:

```
var person = new Person(id);

var result = await _someApiClient.GetAsync(id, cancellationToken);

for(var i = 0; i < max; i++)
{
   // i is implicitly typed
}

var person = this.personFactory.GetOrCreate(id);
this.someDependency.SubmitPerson(person);
```

24. Always Use Access Modifiers

Always use access modifiers! I know, the default is sometimes what you want: “private” for members, “internal” for classes. But it really does help to be explicit about this. It shows that you were purposeful in your intent.

And while you’re at it, use the lowest necessary modifier. This is equivalent to the principle of least privilege. For example, don’t use protected when private will do. Use protected internal to give access only to subclasses in the assembly. If your class is public and you want to share a member with a subclass outside the assembly, make the member protected. Here’s an example to clarify.

In my assembly, I make a class to read a certain file type.

```
public FileReader
{
   protected virtual string Read(string location)
   {
      // read file and return result
   }
}
```

You reference my assembly or use NuGet to add it to your project. Since the Read method is protected and virtual, you can not only access it from your own subclass, but you can override it.

25. Always declare the properties as private so as to achieve Encapsulation and ensure data hiding.

```
// Correct
private int employeeId { get; set; }

// Avoid
public int employeeId { get; set; }
```

26. Use Auto Properties

An auto property creates a backing field for you; you just can't see it or access it. If you don't need control over the backing field, don't create one. Instead, just use the auto property (the shortcut is "prop"). It'll save a lot of extra code!

```
public int Count { get; set; }
```

This is an example of an auto property. It makes your code much cleaner.