20 Important Tips To Write Clean C# Code – MUST SHARE

By A.K.Raju

June 14, 2023

Writing code is not difficult. Writing Clean and Scalable Code is not easy as well. In this article, we will talk about **20 Tips to write Clean C# Code** for your projects. In a conversation with a senior, she said 'Your codebase should look like a storybook.' That's a quite simple way to put things. What it means is that, at first glance, any developer that has never seen your code before, has to understand it as much as possible. Each class should tell a story. This helps quite a lot when working in bigger teams. It helps you understand the code better.

Trust me, there were times when I looked at my own code and couldn't figure the heads and tails of it. I bet that this is something you would have faced as well. Writing Clean C# Code is a practice that you would never leave once you get started.

Here is a List of Important Tips to Write Clean C# Code.

1. Use a Good IDE

First things first, Choose the Best IDE Available for your Tech Stack. In our case, Visual Studio is one of the most popular and betters IDEs for C#. It is a solid and completely FREE product of Microsoft. Some developers prefer Rider IDE as well (paid). Working with these IDEs makes sure that your code remains tidy. Visual Studio has pretty stable Intellisense features that can correct and suggest changes in code. You can find a detailed tutorial on how to install Visual Studio 2019 Community here.

2. Use Meaningful Names

Naming Variables is probably the hardest part of the entire development cycle 😛
Thinking of meaningful names for your variables and methods is quite time-consuming. But skipping this process and giving random names is not a good idea either, is it?

Bad Practice

```
int d;
```

This is the easiest way to name variables right? But DO NOT do it. A good name helps other developers to understand the context and usage of the variable/method. Here is how you would want to name your variables.

Good Practice

```
int daysToAppocalypse;
```

3. Use Camel/Pascal Case Notation

Apart from choosing an appropriate name for your variables, also maintain how you write the names. Ideally, we use Camel Case and Pascal Case Notation as best code practices. Do not use random Capitals throughout your variables. **That just doesn't look pretty!**

Camel Case Notation

Basically, the first letter of the first word of the variable will be in lower case, and the first letter of every other word that follows should be in upper case. You will have to use this notation while naming your local variables and method arguments.

Bad Practice

```
int RandomInteger;
string FirstName;
```

Good Practice

```
int randomInteger;
string firstName;
```

Pascal Case Notation

Here, the first letters of all your words should be in Upper Case. We use this kind of notation for naming Methods and Classes

Bad Practice

```
class program
{
    static void main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Good Practice

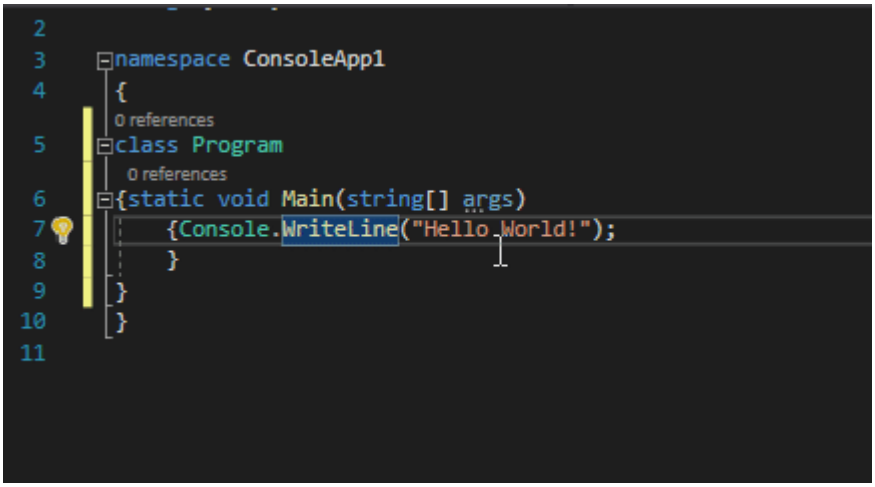
```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

4. Pay attention to Formatting

Formatting your code improves code readability. Tabs over Spaces, Remember?

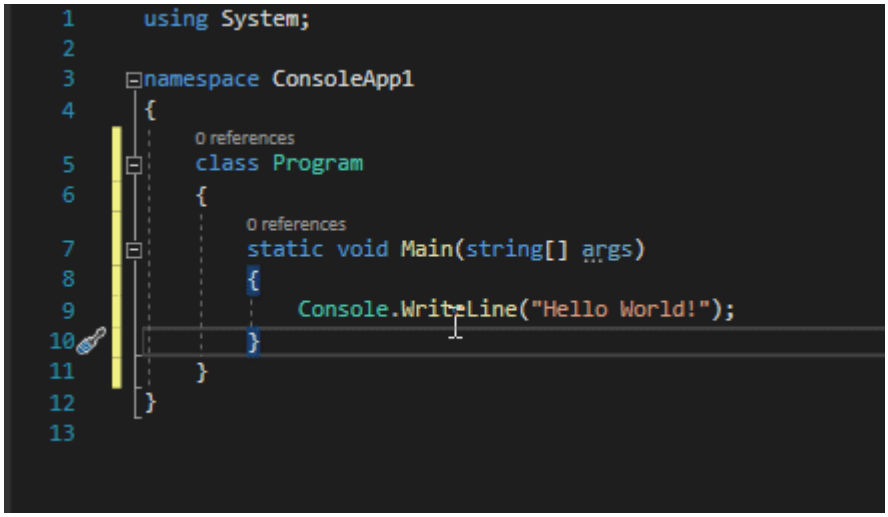
```
class Program
{static void Main(string[] args)
    {Console.WriteLine("Hello World!");
    }
}
```

How does this look? Pretty annoying, yeah? Now, Visual Studio has a built-in feature to format your code perfectly. To do this, go to the concerned class and simply Press CTRL + K and CTRL + D. See it? Cool, yeah?

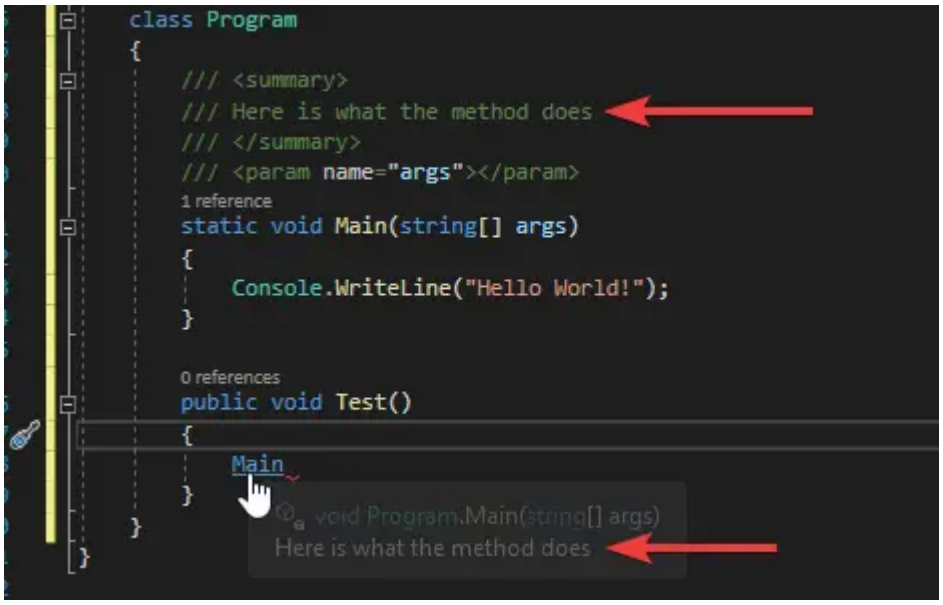


5. Add Comments Whenever Neccesary

This is something we all developers Hate, don't we? 😀 However, adding a few lines of comments/description of what the method actually does helps you and the other developers in the longer run. Visual Studio makes it much easy for you. Simply Go above the concerned method and type in ///, VS automatically generates a comment template for you including the arguments of the method.



Now, why is this cool? Whenever you call this method (from anywhere), Visual Studio shows your comment too. Trust me, it is very helpful.



PS, Add comments only when the situations demands you to. For example, when a particlar method is too complex and requires in-depth explanation. This is one case where your would like to add comments. Do remember that maintaining comments will become a task as well. So use comments sparingly.

6. Reuse Code

Writing codes that can be reusable is quite vital. It can decrease the overall lines of code in your project and makes it highly efficient. You do not want to copy-paste a function through multiple classes. Rather what you could do is, make a Shared Library Project and reference it in each of the required projects. This way, we build reusable functions. And, if there is any modification needed, you would just have to change the code in the shared library, not everywhere.

7. Keep Class Size Small

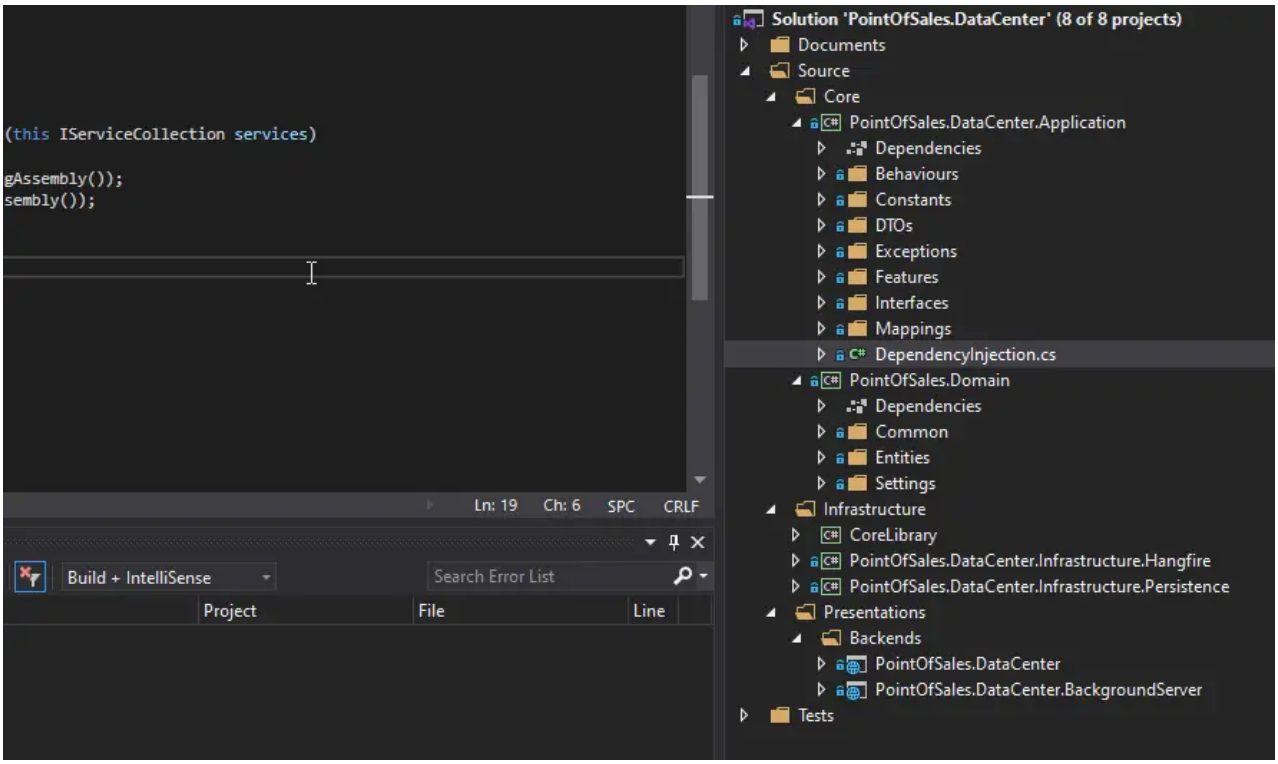
According to the Solid Principles, you must segregate classes to small blocks which has a single responsibilty function only. This helps us to acheive loosely coupled code. Make sure that you dont need to scroll over and over while viewing a class. This can be a general rule of thumb.

8. Use Design Patterns

This may be something to do for an architect level developer. It takes quite a lot of expeirience to determine what kind of design patterns to apply to which scenario. Design patterns are basically patterns that can provide a resuable solution while architecting solutions.

9. Structure your Solution well

Do you build Structured Solutions? It is highly satisfying and important to build one. Here is one of my on-going Solutions following an Onion Architecture.



You understand the point. It is still possible to do everything within a single project. But, in order to favor scalability and loosely couple the solutions, we split them up to various layers like Application, Domain, Infrastructure, and so on.

Here are a few other advantages as well.
1. Reusability – If you want to use the same Project for another solution, you could do so.
2. Improved Security
3.Higly Maintainable
4.Scalable
5.Inversion of controls, etc

> I am planning to write a detailed article on Clean Architecture for ASP.NET Core and MVC Applications later. Let me know if you would be interested in such an article in this comments section below! 😀
>
> Update (20-June-2020) : I have recently published an article on Onion Architecture, that is considered to be one of the cleanest ways to build and maintain solutions. I have demonstrated the architecture along with CRRS and other features .(Source code included too) – Read the blog post here.

10. Avoid Magic Strings/Numbers

What are Magic Strings? They are strings that are directly specified within the application code which has a direct impact on the application behavior. In other words, do not use hardcoded strings or values in our application. It would be tough to keep track of such strings when your application grows. Also, these strings can be associated with some

kind of external references, like a file name, file path, URL, etc. In such cases, when then the location of the resources change, all these magic strings would have to be updated, else that application breaks. Consider the example

```
if(userRole == "Admin")
{
    //logic here
}
```

Instead of this, you could

```
const string ADMIN_ROLE = "Admin"
if(userRole == ADMIN_ROLE )
{
    //logic here
}
```

Alternatively,you could also create an Enum for the User Roles and simply use it. This a much cleaner way to write code.

11. Remove Unused Code

There is often a practice of commenting out the unused code. This ultimately increases the lines of code when the application gets compiled. You do not want to do this. You could use source controls like Git to make sure that you can revert back at any time. Prefer using Git over commenting out code.

12.Use Method Chaining

This is a common technique used extensively in the default generated codes by Microsoft. Here, each method returns an object and these functions will be chained together in a single line. Recognize it? This is a good example of method chaining.

```
services.AddHealthChecks().AddSqlServer(_configuration.GetConnectionString("Default
```

Here is a detailed example. We have a student class. And another random method that creates and returns a data filled student object.

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public Student SomeMethod()
{
    Student testStudent = new Student();
    testStudent.Name = "Mukesh Murugan";
    testStudent.Age = 25;
    return testStudent;
}
```

Setting the values to the Student object may not be an issue for us developers. But, let's say a unit test developer has to test on your class and really doesn't get C#, or you want to please your client by making this whole process simple. This is where Fluent Interfaces come in. Create a new Fluent class like below.

```
public class StudentFluent
{
    private Student student = new Student();
    public StudentFluent AddName(string name)
    {
        student.Name = name;
        return this;
    }
    public StudentFluent AddAge(int age)
    {
        student.Age = age;
        return this;
    }
}

public StudentFluent SomeMethod()
{
    return new StudentFluent().AddName("Mukesh Murugan").AddAge(25);
}
```

This makes so much sense and improves the readability to a whole new level, right? Another simpler example of method chaining is as follows.

```
public string AnotherMethod()
{
    string name = "Mukesh Murugan";
    return name.Replace("M", "A").Replace("A", "M").Replace(".", string.Empty);
}
```

13.Use Async/Await

Asynchronous programming is the way to go! Asynchronous Programming helps improve the overall efficiency while dealing with functions that can take some time to finish computing. During such function executions, the complete application may seem to be frozen to the end-user. This results in bad user experience. In such cases, we use async methods to free the main thread.

Read more here.

14. Don't use 'throw ex' in the catch block

You really don't want to just 'throw ex' the exception after catching it and lose the stack trace data. Just use 'throw'. By using this, you would be able to store the stack trace as well, which is kind of vital for diagnostics purposes.

Bad Practice

```
try
{
    // Do something..
}
catch (Exception ex)
{
    throw ex;
}
```

Good Practice

```
try
{
    // Do something..
}
catch (Exception ex)
{
    throw;
}
```

15. Use Ternary Operator

Consider the following example. I am sure that many of you still follow this practice.

```
public string SomeMethod(int value)
{
    if(value == 10)
    {
        return "Value is 10";
    }
    else
    {
        return "Value is not 10";
    }
}
```

But what if there is a better and cleaner way? **Introducing Ternary Operators**.

> The conditional **operator** ?: , also known as the **ternary** conditional **operator**, evaluates a Boolean expression and returns the result of one of the two expressions, depending on whether the Boolean expression evaluates to true or false.
>
> *Stolen Shamelessly from docs.microsoft.com 😀 :p Read more here.*

Now the multiple lines of code we wrote earlier can be shrunk to just one line using the ternary operators. You can start imagining how many lines of code this is going to save!

```
public string SomeMethod(int value)
{
    return value == 10 ? "Value is 10" : "Value is not 10";
}
```

16. Use Null Coalescing Operator

Similarly, we have yet another operator that can come in handy while you do null checks. ?? operator is known as Null Coalescing Operator in C#. Read more here.

Consider another example. Here is a small function that takes in a Student object as parameter and checks for the null object. If null, return a new Object with data, else return the same object.

```
public Student SomeMethod(Student student)
{
    if (student != null)
    {
        return student;
    }
    else
    {
        return new Student() { Name = "Mukesh Murugan" };
    }
}
```

Let's add the operator and shrink this function too!

```
public Student SomeMethod(Student student)
{
    return student ?? new Student() { Name = "Mukesh Murugan" };
}
```

17.Prefer String Interpolation

Everytime you want to add dynamic values to strings,we preffered Composite formatting or simply adding them with a plus operator.

```
public string SomeMethod(Student student)
{
    return "Student Name is " + student.Name + ". Age is " + student.Age;
}
```

Starting with C# 6, the String Interpolation feature was introduced. This provides a more readable and cool syntax to create formatted strings. Here is how you use interpolated strings.

```
public string SomeMethod(Student student)
{
    return $"Student Name is {student.Name}. Age is {student.Age}";
}
```

There are quite of features available for String Interpolation .Read them here.

18. Use Expression Bodied Methods

Such methods are used in scenarios where the method body is much smaller than even the method definition itself. Why waste Brackets and lines of code, right? Here is how you would write a Expression Bodied Method.

```
public string Message() => "Hello World!";
```

Read more about Expression Bodied Methods here.

19.Avoid Too Many Parameters

Too many parameters is always a nightmare. If you tend to have more than 3 parameter inputs to any method, why not wrap it into a request object or something and then pass? Let's see a small example.

```
public Student SomeMethod(string name, string city, int age, string section,
DateTime dateOfBirth)
{
    return new Student()
    {
        Age = age,
        Name = name,
        //Other parameters too
    };
}
```

You would probably want it to be like this. Get the idea?

```
public Student SomeMethod(Student student)
{
    return student;
}
```

20.Don't ignore caught errors

This is something I kept on doing. There is a good chance that many of you too do this. We add a try catch block and just ignore the error handling, right? It's a good practice to handle such errors and log it to a table or disk.

```
public void SomeMethod()
{
    try
    {
        DoSomething();
    }
    catch
    {

    }
}
```

```
public void SomeMethod()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        LogItSomewhere(ex);
    }
}
```

Talking about logging, I have written a detailed article on Integrating Serilog (Probably the best Logging Framework for .NET Core) with ASP.NET Core Applications. It's totally feature-packed. Check out the article here.

Practice to Write Clean C# Code

In this article, we have seen quite a few tips to help write better and cleaner c# code. Do you have any suggestions or additions? Comment below and let's expand this list and help other developers in the process as well. Do not forget to share this within your community of developers. Thanks and Happy Coding 😀