THE C# CODING STANDARD

HASSAN HABIB

Guidelines for C# Coding Styles

The C# Coding Standard

Hassan Habib

Free to read online at hassanhabib.com/books/csharp-coding-standard

Thank you, Lord for all you taught me. Forgive me if I forget anything

Contents

1	Files
2	Variables
3	Methods
4	Classes
5	Comments and Documentation

CHAPTER 1

Files

The following are the naming conventions and guidance for naming C# files.

Naming

File names should follow the PascalCase convention followed by the file extension `.cs`.

✓ DO

```
Student.cs
```

✓ ALSO, DO

```
StudentService.cs
```

✗ DON'T

```
student.cs
```

✗ ALSO, DON'T

```
studentService.cs
```

✗ ALSO, DON'T

```
Student_Service.cs
```

Partial Class Files

Partial class files are files that contain nested classes for a root file. For instance:

- StudentService.cs
  - StudentService.Validations.cs
  - StudentService.Exceptions.cs

Both validations and exceptions are partial classes to display a different aspect of any given class in a multi-dimensional space.

✓ DO

```
StudentService.Validations.cs
```

✓ ALSO, DO

```
StudentService.Validations.Add.cs
```

✗ DON'T

```
StudentServiceValidations.cs
```

✗ ALSO, DON'T

```
StudentService_Validations.cs
```

CHAPTER 2

Variables

Naming

Variable names should be concise and representative of nature and the quantity of the value it holds or will potentially hold.

Names

✓ DO

```
var student = new Student();
```

✗ DON'T

```
var s = new Student();
```

✗ ALSO, DON'T

```
var stdnt = new Student();
```

The same rule applies to lambda expressions:

✓ DO

```
students.Where(student => student ... );
```

✗ DON'T

```
students.Where(s => s ... );
```

Plurals

✓ DO

```
var students = new List<Student>();
```

✗ DON'T

```
var studentList = new List<Student>();
```

Names with Types

✓ DO

```
var student = new Student();
```

✗ DON'T

```
var studentModel = new Student();
```

✗ ALSO, DON'T

```
var studentObj = new Student();
```

Nulls or Defaults

If a variable value is its default such as `0` for `int` or `null` for strings and you are not planning on changing that value (for testing purposes for instance) then the name should identify that value.

✓ DO

```
Student noStudent = null;
```

✗ DON'T

```
Student student = null;
```

✓ ALSO, DO

```
int noChangeCount = 0;
```

✗ BUT, DON'T

```
int changeCount = 0;
```

Declarations

Declaring a variable and instantiating it should indicate the immediate type of the variable, even if the value is to be determined later.

Clear Types

If the right side type is clear, then use `var` to declare your variable

✓ DO

```
var student = new Student();
```

✗ DON'T

```
Student student = new Student();
```

Semi-Clear Types

If the right side isn't clear (but known) of the returned value type, then you must explicitly declare your variable with its type.

✓ DO

```
Student student = GetStudent();
```

✗ DON'T

```
var student = GetStudent();
```

Unclear Types

If the right side isn't clear and unknown (such as an anonymous types) of the returned value type, you may use `var` as your variable type.

✓ DO

```
var student = new
{
    Name = "Hassan",
    Score = 100
};
```

Single-Property Types

Assign properties directly if you are declaring a type with one property.

✓ DO

```
var inputStudentEvent = new StudentEvent();
inputStudentEvent.Student = inputProcessedStudent;
```

✗ DON'T

```
var inputStudentEvent = new StudentEvent
{
```

```
    Student = inputProcessedStudent
};
```

✓ ALSO, DO

```
var studentEvent = new StudentEvent
{
    Student = someStudent,
    Date = someDate
};
```

✗ DON'T

```
var studentEvent = new StudentEvent();
studentEvent.Student = someStudent;
studentEvent.Date = someDate;
```

Organization

Breakdown

If a variable declaration exceeds 120 characters, break it down starting from the equal sign.

✓ DO

```
List<Student> washingtonSchoolsStudentsWithGrades =
    await GetAllWashingtonSchoolsStudentsWithTheirGradesAsync();
```

✗ DON'T

```
List<Student> washingtonSchoolsStudentsWithGrades = await
GetAllWashingtonSchoolsStudentsWithTheirGradesAsync();
```

Multiple Declarations

Declarations that occupy two lines or more should have a new line before and after them to separate them from previous and next variables declarations.

✓ DO

```
Student student = GetStudent();

List<Student> washingtonSchoolsStudentsWithGrades =
    await GetAllWashingtonSchoolsStudentsWithTheirGradesAsync();

School school = await GetSchoolAsync();
```

✗ DON'T

```
Student student = GetStudent();
List<Student> washingtonSchoolsStudentsWithGrades =
    await GetAllWashingtonSchoolsStudentsWithTheirGradesAsync();
School school = await GetSchoolAsync();
```

Also, declarations of variables that are of only one line should have no new lines between them.

✓ DO

```
Student student = GetStudent();
School school = await GetSchoolAsync();
```

✗ DON'T

```
Student student = GetStudent();

School school = await GetSchoolAsync();
```

CHAPTER 3

Methods

Naming

Method names should be a summary of what the method is doing, it needs to stay precise and short and representative of the operation with respect to synchrony.

Verbs

Method names must contain verbs in them to represent the action it performs.

✓ DO

```
public List<Student> GetStudents()
{
    ...
}
```

✗ DON'T

```
public List<Student> Students()
{
    ...
}
```

Asynchronousy

Asynchronous methods should be postfixed by the term `Async` such as methods returning `Task` or `ValueTask` in general.

✓ DO

```
public async ValueTask<List<Student>> GetStudentsAsync()
{
    ...
}
```

✗ DON'T

```
public async ValueTask<List<Student>> GetStudents()
{
```

```
    ...
}
```

Input Parameters

Input parameters should be explicit about what property of an object they will be assigned to, or will be used for any action such as search.

✓ DO

```
public async ValueTask<Student> GetStudentByNameAsync(string studentName)
{
    ...
}
```

✗ DON'T

```
public async ValueTask<Student> GetStudentByNameAsync(string text)
{
    ...
}
```

✗ ALSO, DON'T

```
public async ValueTask<Student> GetStudentByNameAsync(string name)
{
    ...
}
```

Action Parameters

If your method is performing an action with a particular parameter specify it.

✓ DO

```
public async ValueTask<Student> GetStudentByIdAsync(Guid studentId)
{
    ...
}
```

✗ DON'T

```
public async ValueTask<Student> GetStudentAsync(Guid studentId)
{
    ...
}
```

Passing Parameters

When utilizing a method, if the input parameters aliases match the passed in variables in part or in full, then you don't have to use the aliases, otherwise you must specify your values with aliases.

Assume you have a method:

```
ValueTask<Student> GetStudentByNameAsync(string studentName);
```

✓ DO

```
string studentName = "Todd";
Student student = await GetStudentByNameAsync(studentName);
```

✓ ALSO, DO

```
Student student = await GetStudentByNameAsync(studentName: "Todd");
```

✓ ALSO, DO

```
Student student = await GetStudentByNameAsync(toddName);
```

✗ DON'T

```
Student student = await GetStudentByNameAsync("Todd");
```

✗ DON'T

```
Student student = await GetStudentByNameAsync(todd);
```

Organization

In general encapsulate multiple lines of the same logic into their own method, and keep your method at level 0 of details at all times.

Single/Multiple-Liners

One-Liners

Any method that contains only one line of code should use fat arrows

✓ DO

```
public List<Student> GetStudents() => this.storageBroker.GetStudents();
```

✗ DON'T

```
public List<Student> GetStudents()
{
    return this.storageBroker.GetStudents();
}
```

If a one-liner method exceeds the length of 120 characters then break after the fat arrow with an extra tab for the new line.

✓ DO

```
public async ValueTask<List<Student>> GetAllWashingtonSchoolsStudentsAsync() =>
    await this.storageBroker.GetStudentsAsync();
```

✗ DON'T

```
public async ValueTask<List<Student>> GetAllWashingtonSchoolsStudentsAsync() =>
await this.storageBroker.GetStudentsAsync();
```

Multiple-Liners

If a method contains multiple liners separated or connected via chaining it must have a scope. Unless the parameters are going on the next line then a one-liner method with multi-liner params is allowed.

✓ DO

```
public Student AddStudent(Student student)
{
    ValidateStudent(student);

    return this.storageBroker.InsertStudent(student);
}
```

✓ ALSO, DO

```
public Student AddStudent(Student student)
{
    return this.storageBroker.InsertStudent(student)
        .WithLogging();
}
```

✓ ALSO, DO

```
public Student AddStudent(
    Student student)
{
    return this.storageBroker.InsertStudent(student);
}
```

✗ DON'T

```
public Student AddStudent(Student student) =>
    this.storageBroker.InsertStudent(student)
        .WithLogging();
```

✗ ALSO, DON'T

```
public Student AddStudent(
    Student student) =>
        this.storageBroker.InsertStudent(student);
```

Returns

For multi-liner methods, take a new line between the method logic and the final return line (if any).

✓ DO

```
public List<Student> GetStudents()
{
    StudentsClient studentsApiClient = InitializeStudentApiClient();

    return studentsApiClient.GetStudents();
}
```

✗ DON'T

```
public List<Student> GetStudents()
{
    StudentsClient studentsApiClient = InitializeStudentApiClient();
    return studentsApiClient.GetStudents();
}
```

Multiple Calls

With multiple method calls, if both calls are less than 120 characters then they may stack unless the final call is a method return, otherwise separate with a new line.

✓ DO

```
public List<Student> GetStudents()
{
    StudentsClient studentsApiClient = InitializeStudentApiClient();
    List<Student> students = studentsApiClient.GetStudents();

    return students;
}
```

✗ DON'T

```
public List<Student> GetStudents()
{
    StudentsClient studentsApiClient = InitializeStudentApiClient();

    List<Student> students = studentsApiClient.GetStudents();

    return students;
}
```

✓ ALSO, DO

```
public async ValueTask<List<Student>> GetStudentsAsync()
{
    StudentsClient washingtonSchoolsStudentsApiClient =
        await InitializeWashingtonSchoolsStudentsApiClientAsync();

    List<Student> students = washingtonSchoolsStudentsApiClient.GetStudents();

    return students;
}
```

✗ DON'T

```
public async ValueTask<List<Student>> GetStudentsAsync()
{
    StudentsClient washingtonSchoolsStudentsApiClient =
        await InitializeWashingtonSchoolsStudentsApiClientAsync();
    List<Student> students = washingtonSchoolsStudentsApiClient.GetStudents();

    return students;
}
```

Declaration

A method declaration should not be longer than 120 characters.

✓ DO

```
public async ValueTask<List<Student>>
GetAllRegisteredWashingtonSchoolsStudentsAsync(
    StudentsQuery studentsQuery)
{
    ...
}
```

✗ DON'T

```
public async ValueTask<List<Student>>
GetAllRegisteredWashingtonSchoolsStudentsAsync(StudentsQuery studentsQuery)
```

```
{
    ...
}
```

Multiple Parameters

If you are passing multiple parameters, and the length of the method call is over 120 characters, you must break by the parameters, with **one** parameter on each line.

✓ DO

```
List<Student> redmondHighStudents = await
QueryAllWashingtonStudentsByScoreAndSchoolAsync(
    minimumScore: 130,
    schoolName: "Redmond High");
```

✗ DON'T

```
List<Student> redmondHighStudents = await
QueryAllWashingtonStudentsByScoreAndSchoolAsync(
    minimumScore: 130,schoolName: "Redmond High");
```

Chaining (Uglification/Beautification)

Some methods offer extensions to call other methods. For instance, you can call a `Select()` method after a `Where()` method. And so on until a full query is completed.

We will follow a process of Uglification Beautification. We uglify our code to beautify our view of chained methods. Here's some examples:

✓ DO

```
    students.Where(student => student.Name is "Elbek")
        .Select(student => student.Name)
            .ToList();
```

✗ DON'T

```
    students
    .Where(student => student.Name is "Elbek")
    .Select(student => student.Name)
    .ToList();
```

The first approach enforces simplifying and cutting the chaining short as more calls continue to uglify the code like this:

```
    students.SomeMethod(...)
        .SomeOtherMethod(...)
            .SomeOtherMethod(...)
                .SomeOtherMethod(...)
                    .SomeOtherMethod(...);
```

The uglification process forces breaking down the chains to smaller lists then processing it. The second approach (no uglification approach) may require additional cognitive resources to distinguish between a new statement and an existing one as follows:

```
    students
    .Where(student => student.Name is "Elbek")
    .Select(student => student.Name)
    .OrderBy(student => student.Name)
    .ToList();
    ProcessStudents(students);
```

CHAPTER 4

Classes

Naming

Classes that represent services or brokers in a Standard-Compliant architecture should represent the type of class in their naming convention, however that doesn't apply to models.

Models

✓ DO

```
class Student {
    ...
}
```

✗ DON'T

```
class StudentModel {

}
```

Services

In a singular fashion, for any class that contains business logic.

✓ DO

```
class StudentService {
    ....
}
```

✗ DON'T

```
class StudentsService{
    ...
}
```

✗ ALSO, DON'T

```
class StudentBusinessLogic {
    ...
}
```

✗ ALSO, DON'T

```
class StudentBL {
    ...
}
```

Brokers

In a singular fashion, for any class that is a shim between your services and external resources.

✓ DO

```
class StudentBroker {
    ....
}
```

✗ DON'T

```
class StudentsBroker {
    ...
}
```

Controllers

In a plural fashion, to reflect endpoints such as `/api/students` to expose your logic via RESTful operations.

✓ DO

```
class StudentsController {
    ....
}
```

✗ DON'T

```
class StudentController {
    ...
}
```

Fields

A field is a variable of any type that is declared directly in a class or struct. Fields are members of their containing type.

Naming

Class fields are named in a camel cased fashion.

✓ DO

```
class StudentsController {
    private readonly string studentName;
}
```

✗ DON'T

```
class StudentsController {
    private readonly string StudentName;
}
```

✗ ALSO, DON'T

```
class StudentsController {
    private readonly string _studentName;
}
```

Should follow the same rules for naming as mentioned in the Variables sections.

Referencing

When referencing a class private field, use `this` keyword to distinguish private class member from a scoped method or constructor level variable.

✓ DO

```
class StudentsController {
    private readonly string studentName;

    public StudentsController(string studentName) {
        this.studentName = studentName;
    }
}
```

✗ DON'T

```
class StudentsController {
    private readonly string _studentName;
```

```
    public StudentsController(string studentName) {
        _studentName = studentName;
    }
}
```

Instantiations

Input Params Aliases

If the input variables names match to input aliases, then use them, otherwise you must use the aliases, especially with values passed in.

✓ DO

```
int score = 150;
string name = "Josh";

var student = new Student(name, score);
```

✓ ALSO, DO

```
var student = new Student(name: "Josh", score: 150);
```

✗ BUT, DON'T

```
var student = new Student("Josh", 150);
```

✗ ALSO, DON'T

```
Student student = new (...);
```

Honoring Property Order

When instantiating a class instance - make sure that your property assignment matches the properties order in the class declarations.

✓ DO

```
public class Student
{
    public Guid Id {get; set;}
    public string Name {get; set;}
}

var student = new Student
{
    Id = Guid.NewGuid(),
```

```
    Name = "Elbek"
};
```

✓ ALSO, DO

```
public class Student
{
    private readonly Guid id;
    private readonly string name;

    public Student(Guid id, string name)
    {
        this.id = id;
        this.name = name;
    }
}

var student = new Student (id: Guid.NewGuid(), name: "Elbek");
```

✗ DON'T

```
public class Student
{
    public Guid Id {get; set;}
    public string Name {get; set;}
}

var student = new Student
{
    Name = "Elbek",
    Id = Guid.NewGuid()
};
```

✗ ALSO, DON'T

```
public class Student
{
    private readonly Guid id;
    private readonly string name;

    public Student(string name, Guid id)
    {
        this.id = id;
        this.name = name;
    }
}

var student = new Student (id: Guid.NewGuid(), name: "Elbek");
```

✗ ALSO, DON'T

```
public class Student
{
    private readonly Guid id;
    private readonly string name;

    public Student(Guid id, string name)
    {
        this.id = id;
        this.name = name;
    }
}

var student = new Student (name: "Elbek", id: Guid.NewGuid());
```

CHAPTER 5

Comments and Documentation

Introduction

Comments can only be used to explain what code can't. Whether the code is visible or not.

Copyrights

Comments highlighting copyrights should follow this pattern:

✓ DO

```
// ------------------------------------------------------------
// Copyright (c) Coalition of the Good-Hearted Engineers
// FREE TO USE TO CONNECT THE WORLD
// ------------------------------------------------------------
```

✗ DON'T

```
//------------------------------------------------------------
// <copyright file="StudentService.cs" company="OpenSource">
//      Copyright (C) Coalition of the Good-Hearted Engineers
// </copyright>
//------------------------------------------------------------
```

✗ ALSO, DON'T

```
/*
 * ============================================================
 * Copyright (c) Coalition of the Good-Hearted Engineers
 * FREE TO USE TO CONNECT THE WORLD
 * ============================================================
 */
```

Methods

Methods that have code that is not accessible at dev-time, or perform a complex function should contain the following details in their documentation.

- Purposing
- Incomes
- Outcomes
- Side Effects