Common C# code conventions

Article • 08/01/2023

A code standard is essential for maintaining code readability, consistency, and collaboration within a development team. Following industry practices and established guidelines helps ensure that code is easier to understand, maintain, and extend. Most projects enforce a consistent style through code conventions. The dotnet/docs and dotnet/samples projects are no exception. In this series of articles, you learn our coding conventions and the tools we use to enforce them. You can take our conventions as-is, or modify them to suit your team's needs.

We chose our conventions based on the following goals:

1. *Correctness*: Our samples are copied and pasted into your applications. We expect that, so we need to make code that's resilient and correct, even after multiple edits.
2. *Teaching*: The purpose of our samples is to teach all of .NET and C#. For that reason, we don't place restrictions on any language feature or API. Instead, those samples teach when a feature is a good choice.
3. *Consistency*: Readers expect a consistent experience across our content. All samples should conform to the same style.
4. *Adoption*: We aggressively update our samples to use new language features. That practice raises awareness of new features, and makes them more familiar to all C# developers.

> **Important**
>
> These guidelines are used by Microsoft to develop samples and documentation. They were adopted from the **.NET Runtime, C# Coding Style** and **C# compiler (roslyn)** guidelines. We chose those guidelines because they have been tested over several years of Open Source development. They've helped community members participate in the runtime and compiler projects. They are meant to be an example of common C# conventions, and not an authoritative list (see **Framework Design Guidelines** for that).
>
> The *teaching* and *adoption* goals are why the docs coding convention differs from the runtime and compiler conventions. Both the runtime and compiler have strict performance metrics for hot paths. Many other applications don't. Our *teaching* goal mandates that we don't prohibit any construct. Instead, samples show when constructs should be used. We update samples more aggressively than most

production applications do. Our *adoption* goal mandates that we show code you should write today, even when code written last year doesn't need changes.

This article explains our guidelines. The guidelines have evolved over time, and you'll find samples that don't follow our guidelines. We welcome PRs that bring those samples into compliance, or issues that draw our attention to samples we should update. Our guidelines are Open Source and we welcome PRs and issues. However, if your submission would change these recommendations, open an issue for discussion first. You're welcome to use our guidelines, or adapt them to your needs.

Tools and analyzers

Tools can help your team enforce your standards. You can enable any of the Code analysis tools to enforce the rules you prefer. You can also create an editorconfig so that Visual Studio automatically enforces your style guidelines. You can start by using the dotnet/docs to use our style as a starting point.

These tools make it easier for your team to adopt your preferred guidelines. Visual Studio applies the rules in all `.editorconfig` files in scope to format your code. You can use multiple rule sets to enforce corporate-wide standards, team standards, and even granular project standards.

Any configured code analysis tools produce warnings and diagnostics when its rules are violated. You configure the rules you want applied to your project. Then, each CI build notifies developers when they violate any of the rules.

Language guidelines

The following sections describe practices that the .NET docs team follows to prepare code examples and samples. In general, follow these practices:

- Utilize modern language features and C# versions whenever possible.
- Avoid obsolete or outdated language constructs.
- Only catch exceptions that can be properly handled; avoid catching generic exceptions.
- Use specific exception types to provide meaningful error messages.
- Use LINQ queries and methods for collection manipulation to improve code readability.
- Use asynchronous programming with async and await for I/O-bound operations.
- Be cautious of deadlocks and use Task.ConfigureAwait when appropriate.

- Use the language keywords for data types instead of the runtime types. For example, use `string` instead of System.String, or `int` instead of System.Int32.
- Use `int` rather than unsigned types. The use of `int` is common throughout C#, and it's easier to interact with other libraries when you use `int`. Exceptions are for documentation specific to unsigned data types.
- Use `var` only when a reader can infer the type from the expression. Readers view our samples on the docs platform. They don't have hover or tool tips that display the type of variables.
- Write code with clarity and simplicity in mind.
- Avoid overly complex and convoluted code logic.

More specific guidelines follow.

String data

- Use string interpolation to concatenate short strings, as shown in the following code.

```
C#
```

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- To append strings in loops, especially when you're working with large amounts of text, use a System.Text.StringBuilder object.

```
C#
```

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Arrays

- Use the concise syntax when you initialize arrays on the declaration line. In the following example, you can't use `var` instead of `string[]`.

```
C#
```

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- If you use explicit instantiation, you can use `var`.

C#

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

Delegates

- Use Func<> and Action<> instead of defining delegate types. In a class, define the delegate method.

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Call the method using the signature defined by the `Func<>` or `Action<>` delegate.

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- If you create instances of a delegate type, use the concise syntax. In a class, define the delegate type and a method that has a matching signature.

C#

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
```

```
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- Create an instance of the delegate type and call it. The following declaration shows the condensed syntax.

```
C#

Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- The following declaration uses the full syntax.

```
C#

Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

`try-catch` and `using` statements in exception handling

- Use a try-catch statement for most exception handling.

```
C#

static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
    }
}
```

- 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.

  In the following example, the `try-finally` statement only calls `Dispose` in the `finally` block.

C#

```
Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    if (bodyStyle != null)
    {
        ((IDisposable)bodyStyle).Dispose();
    }
}
```

You can do the same thing with a `using` statement.

C#

```
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

Use the new using syntax that doesn't require braces:

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

&& and || operators

- Use && instead of & and || instead of | when you perform comparisons, as shown in the following example.

C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
```

```
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

If the divisor is 0, the second clause in the `if` statement would cause a run-time error. But the && operator short-circuits when the first expression is false. That is, it doesn't evaluate the second expression. The & operator would evaluate both, resulting in a run-time error when `divisor` is 0.

`new` operator

- Use one of the concise forms of object instantiation, as shown in the following declarations. The second example shows syntax that is available starting in C# 9.

  C#

  ```
  var firstExample = new ExampleClass();
  ```

  C#

  ```
  ExampleClass instance2 = new();
  ```

  The preceding declarations are equivalent to the following declaration.

  C#

  ```
  ExampleClass secondExample = new ExampleClass();
  ```

- Use object initializers to simplify object creation, as shown in the following example.

  C#

  ```
  var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
      Location = "Redmond", Age = 2.3 };
  ```

  The following example sets the same properties as the preceding example but doesn't use initializers.

  C#

```
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Event handling

- Use a lambda expression to define an event handler that you don't need to remove later:

C#

```
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

The lambda expression shortens the following traditional definition.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Static members

Call static members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Don't qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

LINQ queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers who are located in Seattle.

```
C#
```

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

```
C#
```

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as `Name` and `ID` in the result, rename them to clarify that `Name` is the name of a customer, and `ID` is the ID of a distributor.

```
C#
```

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorID =
distributor.ID };
```

- Use implicit typing in the declaration of query variables and range variables. This guidance on implicit typing in LINQ queries overrides the general rules for implicitly typed local variables. LINQ queries often use projections that create anonymous types. Other query expressions create results with nested generic types. Implicit typed variables are often more readable.

```
C#
```

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Align query clauses under the from clause, as shown in the previous examples.
- Use where clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

C#

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Use multiple `from` clauses instead of a join clause to access inner collections. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

C#

```
var scoreQuery = from student in students
                 from score in student.Scores!
                 where score > 90
                 select new { Last = student.LastName, score };
```

Implicitly typed local variables

- Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment.

C#

```
var message = "This is clearly a string.";
var currentTemperature = 27;
```

- Don't use var when the type isn't apparent from the right side of the assignment. Don't assume the type is clear from a method name. A variable type is considered clear if it's a `new` operator, an explicit cast or assignment to a literal value.

C#

```
int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- Don't use variable names to specify the type of the variable. It might not be correct. Instead, use the type to specify the type, and use the variable name to indicate the semantic information of the variable. The following example should use `string` for the type and something like `iterations` to indicate the meaning of the information read from the console.

  C#

  ```
  var inputInt = Console.ReadLine();
  Console.WriteLine(inputInt);
  ```

- Avoid the use of `var` in place of dynamic. Use `dynamic` when you want run-time type inference. For more information, see Using type dynamic (C# Programming Guide).

- Use implicit typing for the loop variable in for loops.

  The following example uses implicit typing in a `for` statement.

  C#

  ```
  var phrase =
  "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
  var manyPhrases = new StringBuilder();
  for (var i = 0; i < 10000; i++)
  {
      manyPhrases.Append(phrase);
  }
  //Console.WriteLine("tra" + manyPhrases);
  ```

- Don't use implicit typing to determine the type of the loop variable in foreach loops. In most cases, the type of elements in the collection isn't immediately obvious. The collection's name shouldn't be solely relied upon for inferring the type of its elements.

  The following example uses explicit typing in a `foreach` statement.

  C#

  ```
  foreach (char ch in laugh)
  {
      if (ch == 'h')
  ```

```
            Console.Write("H");
        else
            Console.Write(ch);
    }
    Console.WriteLine();
```

- use implicit type for the result sequences in LINQ queries. The section on LINQ explains that many LINQ queries result in anonymous types where implicit types must be used. Other queries result in nested generic types where `var` is more readable.

> **Note**
>
> Be careful not to accidentally change a type of an element of the iterable collection. For example, it is easy to switch from **System.Linq.IQueryable** to **System.Collections.IEnumerable** in a `foreach` statement, which changes the execution of a query.

Some of our samples explain the *natural type* of an expression. Those samples must use `var` so that the compiler picks the natural type. Even though those examples are less obvious, the use of `var` is required for the sample. The text should explain the behavior.

Place the using directives outside the namespace declaration

When a `using` directive is outside a namespace declaration, that imported namespace is its fully qualified name. The fully qualified name is clearer. When the `using` directive is inside the namespace, it could be either relative to that namespace, or its fully qualified name.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
```

```
    }
}
```

Assuming there's a reference (direct, or indirect) to the WaitUntil class.

Now, let's change it slightly:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

And it compiles today. And tomorrow. But then sometime next week the preceding (untouched) code fails with two errors:

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

One of the dependencies has introduced this class in a namespace then ends with `.Azure`:

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

A `using` directive placed inside a namespace is context-sensitive and complicates name resolution. In this example, it's the first namespace that it finds.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Adding a new namespace that matches either `CoolStuff.Azure` or `CoolStuff.AwesomeFeature.Azure` would match before the global `Azure` namespace. You could resolve it by adding the `global::` modifier to the `using` declaration. However, it's easier to place `using` declarations outside the namespace instead.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Style guidelines

In general, use the following format for code samples:

- Use four spaces for indentation. Don't use tabs.
- Align code consistently to improve readability.
- Limit lines to 65 characters to enhance code readability on docs, especially on mobile screens.
- Break long statements into multiple lines to improve clarity.
- Use the "Allman" style for braces: open and closing brace its own new line. Braces line up with current indentation level.
- Line breaks should occur before binary operators, if necessary.

Comment style

- Use single-line comments (`//`) for brief explanations.
- Avoid multi-line comments (`/* */`) for longer explanations. Comments aren't localized. Instead, longer explanations are in the companion article.
- For describing methods, classes, fields, and all public members use XML comments.
- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

```
C#
```

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

Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see Options, Text Editor, C#, Formatting.
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines aren't indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
C#
```

```
if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

Exceptions are when the sample explains operator or expression precedence.

Security

Follow the guidelines in Secure Coding Guidelines.