

Convenciones de código de C# comunes

Artículo • 17/08/2023

Un estándar de código es esencial para mantener la legibilidad, la coherencia y la colaboración del código dentro de un equipo de desarrollo. El código que sigue las prácticas y directrices establecidas en el sector es más fácil de comprender, mantener y ampliar. La mayoría de los proyectos aplican un estilo coherente a través de convenciones de código. Los proyectos [dotnet/docs](#) y [dotnet/samples](#) no son una excepción. En esta serie de artículos, aprenderá nuestras convenciones de codificación y las herramientas que usamos para aplicarlas. Puede tomar nuestras convenciones tal como están o modificarlas para satisfacer las necesidades de su equipo.

Elegimos nuestras convenciones en función de los siguientes objetivos:

1. *Corrección*: nuestros ejemplos se copian y pegan en las aplicaciones. Esperamos que, por lo tanto, debemos hacer que el código sea resistente y correcto, incluso después de varias modificaciones.
2. *Enseñanza*: el propósito de nuestras muestras es enseñar a todos los objetos .NET y C#. Por ese motivo, no se aplican restricciones a ninguna característica de lenguaje o API. En su lugar, esos ejemplos enseñan cuando una característica es una buena opción.
3. *Coherencia*: los lectores esperan una experiencia coherente en todo nuestro contenido. Todos los ejemplos deben ajustarse al mismo estilo.
4. *Adopción*: actualizamos de forma agresiva nuestros ejemplos para usar nuevas características de lenguaje. Esa práctica genera conciencia de las nuevas características y hace que sean más familiares para todos los desarrolladores de C#.

Importante

Microsoft usa las instrucciones para desarrollar ejemplos y documentación. Se adoptaron a partir de las instrucciones de [estilo de codificación de .NET Runtime y C#](#) y guía de [compilador C# \(roslyn\)](#). Hemos elegido esas directrices porque se han probado durante varios años de desarrollo de código abierto. Han ayudado a los miembros de la comunidad a participar en los proyectos de tiempo de ejecución y compilador. Están diseñados para ser un ejemplo de convenciones comunes de C# y no una lista autoritativa (consulte [Directrices de diseño del marco](#) para ello).

Los objetivos de *enseñanza* y *adopción* son los motivos por los que la convención de codificación de documentos difiere de las convenciones del entorno de

ejecución y del compilador. Tanto el tiempo de ejecución como el compilador tienen métricas de rendimiento estrictas para las rutas de acceso activas. Muchas otras aplicaciones no. Nuestro objetivo de *enseñanza* exige que no prohibimos ninguna construcción. En su lugar, los ejemplos muestran cuándo se deben usar construcciones. Actualizamos los ejemplos de forma más agresiva que la mayoría de las aplicaciones de producción. Nuestro objetivo de *adopción* exige que se muestre código que debe escribir hoy, incluso cuando el código escrito el año pasado no necesita cambios.

En este artículo se explica nuestra guía. Las directrices han evolucionado con el tiempo y encontrará ejemplos que no siguen nuestras directrices. Agradecemos las solicitudes de incorporación de cambios que incluyan esos ejemplos en cumplimiento o incidencias que centran nuestra atención en las muestras que debemos actualizar. Nuestras directrices son de código abierto y agradecemos las solicitudes de incorporación de cambios y de incidencias. Sin embargo, si su envío cambiaría estas recomendaciones, abra primero una incidencia para discutirlo. Le animamos a usar nuestras directrices o adaptarlas a sus necesidades.

Herramientas y analizadores

Las herramientas pueden ayudar a su equipo a aplicar sus estándares. Puede habilitar el [análisis de código](#) para aplicar las reglas que prefiera. También puede crear una [editorconfig](#) para que Visual Studio aplique automáticamente las directrices de estilo. Como punto de partida, puede copiar el [archivo del repositorio dotnet/docs](#) para usar nuestro estilo.

Estas herramientas facilitan a su equipo adoptar sus directrices preferidas. Visual Studio aplica las reglas de todos los archivos de `.editorconfig` en el ámbito para dar formato al código. Puede usar varias configuraciones para aplicar estándares corporativos, estándares de equipo e incluso estándares de proyecto pormenorizados.

El análisis de código genera advertencias y diagnósticos cuando se infringen las reglas habilitadas. Configure las reglas que desea aplicar al proyecto. A continuación, cada compilación de CI notifica a los desarrolladores cuando infringen cualquiera de las reglas.

Id. de diagnóstico

- [Elegir los identificadores de diagnóstico adecuados](#) al compilar sus propios analizadores

Convenciones de lenguaje

En las secciones siguientes se describen las prácticas que sigue el equipo de documentación de .NET para preparar las muestras y ejemplos de código. En general, siga estos procedimientos:

- Use las características modernas del lenguaje y las versiones de C# siempre que sea posible.
- Evite construcciones de lenguaje obsoletas.
- Solo trabaje con las excepciones que se pueden controlar correctamente; evite trabajar sobre excepciones genéricas.
- Use tipos de excepción específicos para proporcionar mensajes de error significativos.
- Use consultas y métodos LINQ para la manipulación de recopilación para mejorar la legibilidad del código.
- Use una programación asíncrona con `async` y `await` para operaciones enlazadas a E/S.
- Tenga cuidado con los interbloqueos y use [Task.ConfigureAwait](#) cuando corresponda.
- Use las palabras clave del lenguaje para los tipos de datos en lugar de los tipos de tiempo de ejecución. Por ejemplo, use `string` en vez de [System.String](#) o `int` en lugar de [System.Int32](#).
- Utilice `int` en lugar de tipos sin signo. El uso de `int` es común en todo C#, y es más fácil interactuar con otras bibliotecas cuando se usa `int`. Las excepciones son para la documentación específica de los tipos de datos sin firmar.
- Use `var` solo cuando un lector pueda deducir el tipo de la expresión. Los lectores ven nuestros ejemplos en la plataforma de documentos. No tienen sugerencias pasando el ratón por encima ni las herramientas que muestran el tipo de variables.
- Escriba el código pensando en su claridad y simplicidad.
- Evite la lógica de código demasiado compleja y enrevesada.

Se siguen instrucciones más específicas.

Datos de cadena

- Use [interpolación de cadenas](#) para concatenar cadenas cortas, como se muestra en el código siguiente.

C#

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

```
{nameList[n].FirstName}";
```

- Para anexas cadenas en bucles, especialmente cuando se trabaja con grandes cantidades de texto, utilice un objeto [System.Text.StringBuilder](#).

C#

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

Matrices

- Utilice sintaxis concisa para inicializar las matrices en la línea de declaración. En el siguiente ejemplo, no puede utilizar `var` en lugar de `string[]`.

C#

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

- Si usa la creación de instancias explícita, puede usar `var`.

C#

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

Delegados

- Use `Func<>` y `Action<>` en lugar de definir tipos de delegado. En una clase, defina el método delegado.

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;
```

- Llame al método con la signatura definida por el delegado `Func<>` o `Action<>`.

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)}");
```

- Si crea instancias de un tipo de delegado, utilice la sintaxis concisa. En una clase, defina el tipo de delegado y un método que tenga una firma coincidente.

C#

```
public delegate void Del(string message);  
  
public static void DelMethod(string str)  
{  
    Console.WriteLine("DelMethod argument: {0}", str);  
}
```

- Cree una instancia del tipo de delegado y llámela. La siguiente declaración muestra la sintaxis condensada.

C#

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

- La siguiente declaración utiliza la sintaxis completa.

C#

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

Instrucciones try-catch y using en el control de excepciones

- Use una instrucción [try-catch](#) en la mayoría de casos de control de excepciones.

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;
    }
}
```

- Simplifique el código mediante la [instrucción using](#) de C#. Si tiene una instrucción [try-finally](#) en la que el único código del bloque `finally` es una llamada al método [Dispose](#), use en su lugar una instrucción `using`.

En el ejemplo siguiente, la instrucción `try-finally` solo llama a `Dispose` en el bloque `finally`.

C#

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

Puede hacer lo mismo con una instrucción `using`.

C#

```
using (Font arial = new Font("Arial", 10.0f))
{
```

```
byte charset2 = arial.GdiCharSet;
}
```

Use la nueva [sintaxis using](#) que no requiere corchetes:

C#

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

Operadores && y ||

- Use [&&](#) en vez de [&](#) y [||](#) en vez de [|](#) cuando realice comparaciones, como se muestra en el ejemplo siguiente.

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) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Si el divisor es 0, la segunda cláusula de la instrucción `if` produciría un error en tiempo de ejecución. Pero el operador `&&` cortocircuita cuando la primera expresión es falsa. Es decir, no evalúa la segunda expresión. El operador `&` evaluaría ambos, lo que provocaría un error en tiempo de ejecución cuando `divisor` es 0.

Operador new

- Use una de las formas concisas de creación de instancias de objeto, tal como se muestra en las declaraciones siguientes.

C#

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

C#

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

Las declaraciones anteriores son equivalentes a la siguiente declaración.

C#

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

- Use inicializadores de objeto para simplificar la creación de objetos, tal y como se muestra en el ejemplo siguiente.

C#

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

En el ejemplo siguiente se establecen las mismas propiedades que en el ejemplo anterior, pero no se utilizan inicializadores.

C#

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

Control de eventos

- Use una expresión lambda para definir un controlador de eventos que no es necesario quitar más tarde:

C#

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



```
    };  
}
```

La expresión lambda acorta la siguiente definición tradicional.

C#

```
public Form1()  
{  
    this.Click += new EventHandler(Form1_Click);  
}  
  
void Form1_Click(object? sender, EventArgs e)  
{  
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());  
}
```

Miembros estáticos

Llame a miembros **estáticos** con el nombre de clase: *ClassName.StaticMember*. Esta práctica hace que el código sea más legible al clarificar el acceso estático. No califique un miembro estático definido en una clase base con el nombre de una clase derivada. Mientras el código se compila, su legibilidad se presta a confusión, y puede interrumpirse en el futuro si se agrega a un miembro estático con el mismo nombre a la clase derivada.

Consultas LINQ

- Utilice nombres descriptivos para las variables de consulta. En el ejemplo siguiente, se utiliza `seattleCustomers` para los clientes que se encuentran en Seattle.

C#

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

- Utilice alias para asegurarse de que los nombres de propiedad de tipos anónimos se escriben correctamente con mayúscula o minúscula, usando para ello la grafía Pascal.

C#

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

- Cambie el nombre de las propiedades cuando puedan ser ambiguos en el resultado. Por ejemplo, si la consulta devuelve un nombre de cliente y un identificador de distribuidor, en lugar de dejarlos como `Name` e `ID` en el resultado, cambie su nombre para aclarar que `Name` es el nombre de un cliente e `ID` es el identificador de un distribuidor.

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 };
```

- Utilice tipos implícitos en la declaración de variables de consulta y variables de intervalo. Esta guía sobre la escritura implícita en consultas LINQ invalida las reglas generales de [las variables locales con tipo implícito](#). Las consultas LINQ suelen usar proyecciones que crean tipos anónimos. Otras expresiones de consulta crean resultados con tipos genéricos anidados. Las variables con tipo implícito suelen ser más legibles.

C#

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

- Alinee las cláusulas de consulta bajo la cláusula `from`, como se muestra en los ejemplos anteriores.
- Use cláusulas `where` antes de otras cláusulas de consulta para asegurarse de que las cláusulas de consulta posteriores operan en un conjunto de datos reducido y filtrado.

C#

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

- Use varias cláusulas `from` en lugar de una cláusula `join` para obtener acceso a colecciones internas. Por ejemplo, una colección de objetos `Student` podría contener cada uno un conjunto de resultados de exámenes. Cuando se ejecuta la siguiente consulta, devuelve cada resultado superior a 90, además del apellido del alumno que recibió la puntuación.

C#

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

Variables locales con asignación implícita de tipos

- Use `tipos implícitos` para las variables locales cuando el tipo de la variable sea obvio desde el lado derecho de la tarea.

C#

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

- No use `var` cuando el tipo no sea evidente desde el lado derecho de la tarea. No asuma que el tipo está claro a partir de un nombre de método. Se considera que un tipo de variable es claro si es un operador `new`, una conversión explícita o tarea para un valor literal.

C#

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

- No use nombres de variable para especificar el tipo de la variable. Puede no ser correcto. En su lugar, use el tipo para especificar el tipo y use el nombre de la variable para indicar la información semántica de la variable. En el ejemplo

siguiente se debe usar `string` para el tipo y algo parecido a `iterations` para indicar el significado de la información leída desde la consola.

C#

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

- Evite el uso de `var` en lugar de `dynamic`. Use `dynamic` cuando desee la inferencia de tipos en tiempo de ejecución. Para obtener más información, vea [Uso de tipo dinámico \(Guía de programación de C#\)](#).
- Use la escritura implícita de la variable de bucle en bucles `for`.

En el ejemplo siguiente se usan tipos implícitos en una instrucción `for`.

C#

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

- No use tipos implícitos para determinar el tipo de la variable de bucle en bucles `foreach`. En la mayoría de los casos, el tipo de elementos de la colección no es inmediatamente obvio. El nombre de la colección no debe servir únicamente para inferir el tipo de sus elementos.

En el ejemplo siguiente se usan tipos explícitos en una instrucción `foreach`.

C#

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

- use el tipo implícito para las secuencias de resultados en las consultas LINQ. En la sección sobre [LINQ](#) se explica que muchas consultas LINQ dan lugar a tipos anónimos en los que se deben usar tipos implícitos. Otras consultas dan como resultado tipos genéricos anidados en los que `var` es más legible.

⚠ Nota

Tenga cuidado de no cambiar accidentalmente un tipo de elemento de la colección iterable. Por ejemplo, es fácil cambiar de [System.Linq.IQueryable](#) a [System.Collections.IEnumerable](#) en una instrucción `foreach`, lo cual cambia la ejecución de una consulta.

Algunos de nuestros ejemplos explican *el tipo natural* de una expresión. Esos ejemplos deben usar `var` para que el compilador elija el tipo natural. Aunque esos ejemplos son menos obvios, se requiere el uso de `var` para el ejemplo. El texto debe explicar el comportamiento.

Colocación de directivas `using` fuera de la declaración del espacio de nombres

Cuando una directiva `using` está fuera de la declaración de un espacio de nombres, ese espacio de nombres importado es su nombre completo. El nombre completo es más claro. Cuando la directiva `using` está dentro del espacio de nombres, puede ser relativa al espacio de nombres o su nombre completo.

C#

```
using Azure;

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

Se supone que hay una referencia (directa o indirecta) a la clase [WaitUntil](#).

Ahora vamos a cambiarla ligeramente:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

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

Y se compila hoy. Y mañana. Sin embargo, en algún momento de la próxima semana, el código anterior (sin modificar) produce dos errores:

Consola

- 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

Una de las dependencias ha introducido esta clase en un espacio de nombres y, después, finaliza con `.Azure`:

C#

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

Una directiva `using` colocada dentro de un espacio de nombres es contextual y complica la resolución de nombres. En este ejemplo, es el primer espacio de nombres que encuentra.

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

- Azure

Si se agrega un nuevo espacio de nombres que coincida con `CoolStuff.Azure` o `CoolStuff.AwesomeFeature.Azure`, se tomaría como coincidencia antes que el espacio de nombres global `Azure`. Para resolverlo, agregue el modificador `global::` a la declaración `using`. Sin embargo, es más fácil colocar declaraciones `using` fuera del espacio de nombres.

C#

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

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

Guía de estilo

En general, use el siguiente formato para ejemplos de código:

- Use cuatro espacios para la sangría. No use pestañas.
- Alinee el código de forma coherente para mejorar la legibilidad.
- Limite las líneas a 65 caracteres para mejorar la legibilidad del código en documentos, especialmente en pantallas móviles.
- Divida instrucciones largas en varias líneas para mejorar la claridad.
- Use el estilo "Allman" para llaves: llaves de apertura y cierre en su propia línea nueva. Las llaves se alinean con el nivel de sangría actual.
- Los saltos de línea deben producirse antes de los operadores binarios, si es necesario.

Estilo de comentario

- Use comentarios de una sola línea (`//`) para las explicaciones breves.
- Evite los comentarios de varias líneas (`/* */`) para las explicaciones más largas. Los comentarios no se localizan. En su lugar, las explicaciones más largas se

encuentran en el artículo complementario.

- Para describir métodos, clases, campos y todos los miembros públicos se usan los [comentarios XML](#).
- Coloque el comentario en una línea independiente, no al final de una línea de código.
- Comience el texto del comentario con una letra mayúscula.
- Finalice el texto del comentario con un punto.
- Inserte un espacio entre el delimitador de comentario (//) y el texto del comentario, como se muestra en el ejemplo siguiente.

C#

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

Convenciones de diseño

Un buen diseño utiliza un formato que destaque la estructura del código y haga que el código sea más fácil de leer. Las muestras y ejemplos de Microsoft cumplen las convenciones siguientes:

- Utilice la configuración del Editor de código predeterminada (sangría automática, sangrías de 4 caracteres, tabulaciones guardadas como espacios). Para obtener más información, vea [Opciones, editor de texto, C#, formato](#).
- Escriba solo una instrucción por línea.
- Escriba solo una declaración por línea.
- Si a las líneas de continuación no se les aplica sangría automáticamente, hágalo con una tabulación (cuatro espacios).
- Agregue al menos una línea en blanco entre las definiciones de método y las de propiedad.
- Utilice paréntesis para que las cláusulas de una expresión sean evidentes, como se muestra en el código siguiente.

C#


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

Las excepciones son cuando el ejemplo explica la precedencia de operador o expresión.

Seguridad

Siga las instrucciones de [Instrucciones de codificación segura](#).


Colaborar con nosotros en GitHub


El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)