

Reglas de estilos C#

Código del documento

06/09/2021

Dawid Mateusz

Departamento IT

Contenido

[1. Introducción 4](#_Toc97284924)

[2. Guías de estilo 4](#_Toc97284925)

[2.1 Comentarios 4](#_Toc97284926)

[1.2.1. Tipos de comentarios 4](#_Toc97284927)

[2.2 Sangrías, espacios y líneas en blanco 5](#_Toc97284928)

[1.2.2. Sangrías 5](#_Toc97284929)

[2.2.2. Caracteres espacio 5](#_Toc97284930)

[3.2.2. Líneas en blanco 6](#_Toc97284931)

[2.3 Uso de las llaves 6](#_Toc97284932)

[3. Convección de nombres 7](#_Toc97284933)

[3.1 Enumerados 7](#_Toc97284934)

[3.2 Constantes 7](#_Toc97284935)

[3.3 Miembros inyectados 7](#_Toc97284936)

[1.3.3. Blazor 7](#_Toc97284937)

[2.3.3. Resto de proyectos 7](#_Toc97284938)

[3.4 Métodos 7](#_Toc97284939)

[3.5 Clases 8](#_Toc97284940)

[3.6 Clases abstractas 8](#_Toc97284941)

[3.7 Objetos de transferencia de datos o DTO. 8](#_Toc97284942)

[3.8 Interfaces 8](#_Toc97284943)

[4. Excepciones 9](#_Toc97284944)

[5. Otras recomendaciones 9](#_Toc97284945)

Control Documental

|  |  |  |
| --- | --- | --- |
| Redactado por: |  |  |
| Revisado por: |  |  |
| Aprobado por: |  |  |

Histórico de modificaciones

|  |  |  |  |
| --- | --- | --- | --- |
| CÓDIGO | VERSIÓN | FECHA | COMENTARIO |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introducción

En el presente documento describiremos un conjunto de reglas que nos ayudarán a escribir API’s en ASP.Net Core C#. Con ello intentaremos conseguir que el código tenga las siguientes cualidades:

* Bien organizado
* Fácil de leer
* Fácil de mantener
* Fácil detección de errores
* Eficiente

# Guías de estilo

## Comentarios

Solo se usan para añadir información o para aclarar secciones de código. No se usan para describir el programa. Se añaden junto a la definición de clases, variables, propiedades, métodos, etc. para explicar su propósito, y al comienzo de algunas secciones de código, especialmente complicadas, para explicar su funcionamiento.

En cualquier caso, seguiremos las siguientes reglas generales:

* Los comentarios en general se escriben en líneas que no contienen código y antes del código que queremos clarificar. Esta regla se aplica siempre si el comentario tiene más de una línea.
* Sólo en dos casos se permite poner en la misma línea un comentario y una instrucción: comentarios a una definición de variable, que explica la finalidad de esta variable, y un comentario para indicar final de una estructura del lenguaje

### Tipos de comentarios

Con todo esto se pueden definir los siguientes tipos de comentarios:

* **Comentarios de clase**: Usados en la definición de una clase. Opcionalmente puede contener información acerca de la versión.

Ejemplo:

**/// <summary>Descripción genérica de la clase</summary>**

**///<remarks>Descripción detallada de la clase</remarks>**

* **Comentarios de variables**: En el caso de que una variable requiera comentario éste deberá aparecer al estilo XML. Si únicamente son aclaraciones, se realizarán en la misma línea en que se declara la variable y serán muy breves.

Ejemplo:

**/// <summary>Descripción breve de la variable</summary>**

**private int \_numPruebas; // Contador pruebas realizadas**

* **Comentarios de métodos**: Usados en la definición de los métodos, simplemente describen su funcionalidad.

Ejemplo:

**/// <summary>Descripción breve del método</summary>**

**/// <remarks>Descripción detallada del método</remarks>**

**/// <param name="param1">Descripción parámetro 1</param>**

**/// <returns>Información valor de retorno</returns>**

**/// <exception cref="ex1">Descripción excep1</exception>**

* **Comentarios cortos**: Se usan para describir funcionalidades o datos. Deben situarse en la línea inmediatamente anterior a la funcionalidad que se va a describir. También se usan para indicar el final de una estructura. En caso de utilizar este tipo de comentario, seguir las siguientes reglas:
  + Utilizar uno o más tabuladores para separar la instrucción y el comentario.
  + Si aparece más de un comentario en un bloque de código o bloque de datos, todos comienzan y terminan a la misma altura de tabulación.

Ejemplo:

**// Obtenemos el menor de los datos**

**for (...)**

**{**

**...**

**}**

## Sangrías, espacios y líneas en blanco

Los espacios en blanco facilitan la lectura y el mantenimiento de los programas. Los espacios en blanco que se pueden utilizar son: las sangrías, los caracteres espacio y las líneas en blanco

### Sangrías

El sangrado se utiliza para mostrar la estructura lógica del código. El sangrado óptimo es el formado por cuatro espacios. Es un compromiso entre una estructuración legible y la posibilidad de que alguna línea (con varios sangrados) del código supere el ancho de una línea de una hoja de papel o del monitor. El sangrado se debe realizar con tabulaciones, por tanto, se debe fijar éste en cuatro caracteres.

### Caracteres espacio

Los espacios en blanco sirven para facilitar la lectura de los elementos que forman una expresión. Los espacios en blanco se utilizan en los casos siguientes:

* Las variables y los operadores de una expresión deben estar separados por un elemento en blanco

Ejemplo: Espacio de operadores

**media = suma / cuenta**

* Las listas de definición de variables y las listas de parámetros de una función se debe separar por un espacio en blanco

Ejemplo: Lista de parámetros

**BuscarTickets(int usuario, Datetime fecha)**

### Líneas en blanco

Se utilizan para separar “párrafos” o secciones del código. Cuando leemos un programa entendemos que un fragmento de código entre dos líneas en blanco forma un conjunto con una cierta relación lógica. Veamos como separar secciones o párrafos en un programa:

* Las secciones que forman un programa se separan con al menos una línea en blanco (declaración de constantes, declaración de variables, métodos…).
* Dentro de un subprograma se separan con una línea en blanco los fragmentos de instrucciones muy relacionadas entre sí (por ejemplo, conjunto de instrucciones que realizan una operación).

## Uso de las llaves

Su colocación se debe hacer en líneas reservadas para cada una de ellas, sin ninguna otra instrucción en la línea. Ambas deben ir en la misma columna que la instrucción que la precede.

Ejemplo:

**for (int contador = 0; contador < numIteraciones;**

**contador++)**

**{**

**...**

**}**

Se permiten algunas excepciones a esta norma en las sentencias condicionales y de repetición donde la llave de apertura puede estar al final de la sentencia de control o donde tras la llave de cierre se permiten sentencias condicionales.

Ejemplo: llave al final de la sentencia

**if (contador == 0)**

**{**

**...**

**}**

**else**

**{**

**...**

**}**

Ejemplo: sentencias tras la llave de cierre

**do**

**{**

**...**

**} while (estado != resultOk)**

Se debe evitar en la medida de lo posible omitir las llaves, incluso en aquellos bloques que encierran una única sentencia.

# Convección de nombres

Para elegir los nombres se deben seguir las siguientes recomendaciones generales

* Elegir nombres comprensibles y en relación con la tarea que corresponda al objeto nombrado.
* Utilice palabras completas siempre que sea posible. Las abreviaturas pueden adoptar muchas formas y, en consecuencia, pueden resultar confusas. En el caso en que sea estrictamente necesario usarlas, elija las abreviaturas adecuadamente.
* Cuando se trabaje con siglas o acrónimos, éstos deben mantener todos sus caracteres es mayúsculas.
* Utilizar siempre Notación Pascal

## Variables

Evitar variables públicas en una clase y en su lugar utilizar propiedades.

Para variables privadas globales de la clase con ‘\_’ por delante.

Ejemplo: **private string \_empleadoBanca**;

## Propiedades

Las propiedades deberán capitalizar la primera letra y, si es palabra compuesta, también capitalizando la primera letra de la segunda palabra. Ejemplos:

**public int Edad {private set; get}**

**public string NombreEmpleado {private set; get}**

## Enumerados

Los enumerados o enum tienen que empezar por mayúscula. Ejemplo: **Dias**.

## Constantes

Las constantes tienen que empezar por mayúscula. Ejemplo: **NumeroPi**.

## Miembros inyectados

### Blazor

Los campos o atributos inyectados deben de ser privados y notación Camel.

Ejemplo:

**[Inject]**

**private IJSRuntime js**;

### Resto de proyectos

Los campos o atributos inyectados deben de ser privados, readonly y precedidos por un guion bajo ( \_ ).

Ejemplo: **private readonly IMiServicio\_miServicio**;

## Métodos

En los métodos se debe seguir el estándar verbo-sustantivo para asignar los nombres, donde el verbo ha de estar en infinitivo. Se deberán utilizar al menos dos palabras. Además, debe ser lo suficientemente significativo como para describir la acción que realiza.

Ejemplo: uso del estándar verbo-sustantivo

**public int CalcularEdad ()**

**private int CalcularNominaMensual ()**

Se debe de evitar escribir métodos muy largos en la medida de lo posible, típicamente un método deberá de tener entre 1 y 40 líneas de código, si el método contiene más de 40 líneas de código se deberá de considerar re factorizarlo en métodos separados.

El método deberá de tener solo una tarea o fin, no se debe de combinar tareas en un solo método por más pequeñas que estas sean

También, debemos de indicar obligatoriamente el ámbito del método: private, public, protected, internal o protected internal

Ejemplo:

**// Crea Usuario**

**// Envía un email para proceso de activación de usuario**

**CrearUsuario(usuario);**

**EnviaCorreo(uuidUsuario, correoContenido);**

**void CrearUsuario(DataTable usuario )**

**{**

**// Código para creación de Usuario**

**}**

**void EnviaCorreo ( string uuidUsuario, string correoContenido)**

**{**

**// Envía un correo para informar que se ha guardado el detalle**

**}**

## Clases

Los nombres de las clases deben seguir los conceptos básicos de las funciones. Ejemplo: **RaizCuadrada**

En la medida de lo posible se debe respetar el que haya una única clase por archivo, cuyo nombre será el propio de la clase.

## Clases abstractas

Los nombres de la clase abstracta tienen que cumplir las reglas de las clases básicas y seguidas de la palabra “Base”. Ejemplo: **VehiculoBase**

## Objetos de transferencia de datos o DTO.

Las clases que se utilicen para recibir y enviar información fuera del ámbito del web service. Estas clases irán sucedidas de las siglas DTO. Ejemplo: **PaisesDTO**.

## Interfaces

Las interfaces tienen que ir precedidas de la letra “I” seguida del nombre de la interfaz en mayúscula. Ejemplo: **IVechiculo**

# Excepciones

Para el manejo de las excepciones se debe emplear el mecanismo propio del lenguaje: try -> catch -> finally.

Es conveniente seguir un orden lógico y homogéneo en la recogida de los errores. De este modo, las excepciones en el bloque catch se capturarán en este orden:

* Excepciones de operación
* Excepciones específicas
* Excepciones personalizadas
* Excepciones genéricas

Hay que evitar utilizar try catch globales. Debemos de utilizarlo solamente en la porción de código susceptible a error. De este modo mantendremos el código mas legible y ordenado.

# Otras recomendaciones

* Cuando sea necesario el anidamiento de sentencias condicionales o de repetición, éste no debe superar los cuatro niveles. Si se requieren más se debe considerar la agrupación en funciones.
* Siempre que sea posible se debe reutilizar código existente, ya sean librerías o clases.
* Si la eficiencia es importante (sistemas críticos o de tiempo real) y es necesario utilizar expresiones no comprensibles y muy complicadas pero rápidas, añadir comentarios que ayuden a comprender el código.
* Reducir al mínimo las operaciones de entrada / salida.
* Cuando se pasan estructuras grandes a un subprograma, hacerlo por referencia. Esto evita manipular datos sobre la pila y hace la ejecución más rápida.
* Es importante que los nombres no coincidan con palabras reservadas.
* Si existe la necesidad de pasar más de 5~6 parámetros a un método especifico, este es un buen candidato para definir una clase con parámetros y una estructura de datos tipada, de lo contrario el consumo de memoria es afectado de forma considerable, permite la corrupción de datos, y se castiga en ciclos de procesamiento al servidor lo cual pega en performance.