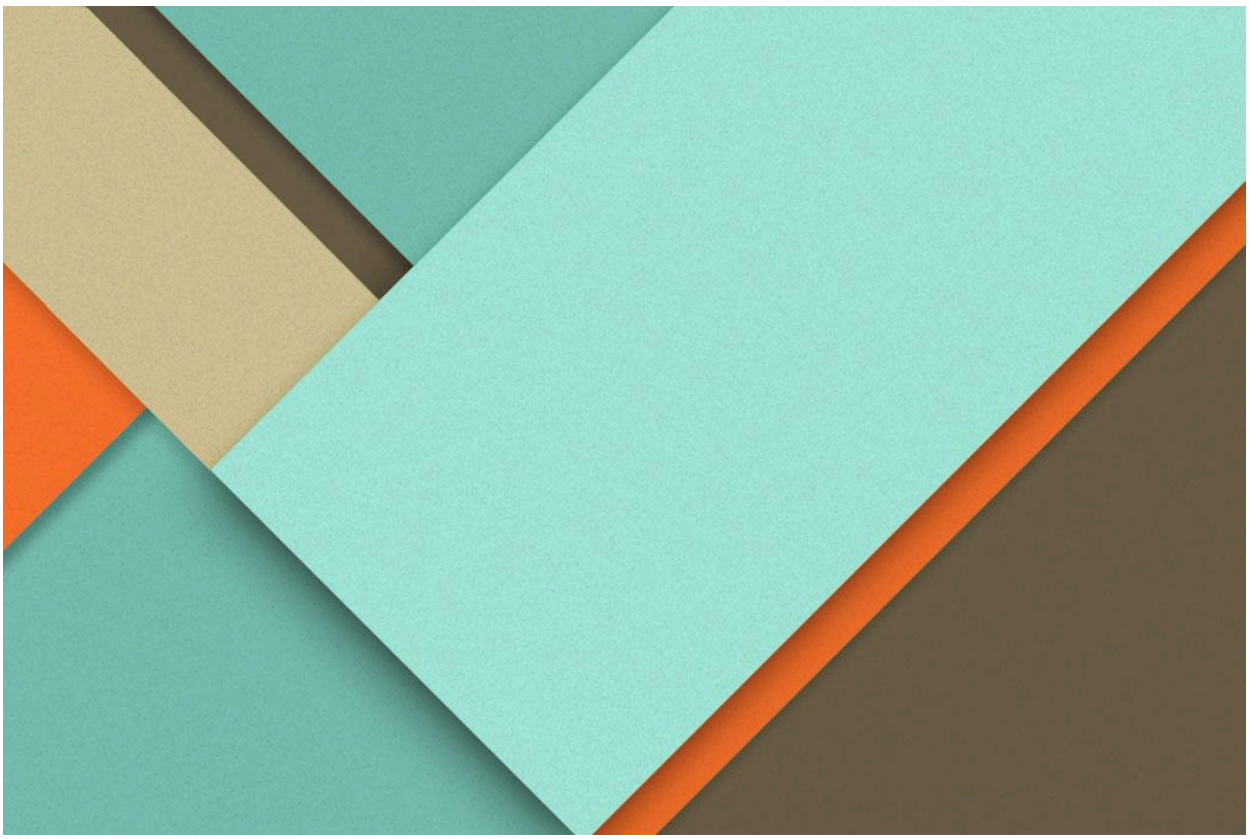

Estándar de codificación para C# y .Net

21/08/2025

Tecnologías para la Construcción de Software

Juan Carlos Pérez Arriaga

Leonardo Daniel Ortega Teoba

Gabriel Antonio González López

Universidad Veracruzana, Facultad de Estadística e Informática, Ingeniería de Software

Contenido

Introducción

Este estándar está dirigido para estudiantes de Tecnologías para la Construcción de Software en la carrera de Ingeniería de Software, se recomienda usarlo para el desarrollo de los proyectos en clase y productos de software, con el objetivo de establecer una base para la construcción y lectura del código, promoviendo buenas prácticas de programación (convenciones del lenguaje brindadas por Microsoft), su legibilidad, orden, espaciado y nombramiento, accesibilidad a su mantenimiento para aquel que busque modificarlo y probarlo para verificar su seguridad y funcionamiento en el desarrollo de este producto de software.

El propósito de este estándar es para darnos un seguimiento en nuestra formación como ingenieros de software siguiendo parámetros definidos desde un comienzo como una guía clara y estructurada para el desarrollo de software apegada completamente a las siguientes pautas que hemos definido.

Propósito

Este estándar tiene como propósito brindar especificaciones que ayuden a desarrollar un código seguro, mantenible y legible con C#.

Se entiende como código seguro a aquel que no sea susceptible a inyecciones de código, acceso a datos sensibles no cifrados o sin una verificación de credenciales previa.

Idioma

Para el nombramiento del código se usará el idioma inglés. Se tomó esta decisión para motivar a los estudiantes a desarrollar y mejorar su dominio del idioma, así como adaptarse al ambiente laboral general del mundo.

En cuanto al lenguaje de la GUI (Graphic User Interface) puede ser en cualquier idioma, la aplicación también puede incluir varios perfiles de idiomas.

Reglas de nombramiento

Absolutamente todos los nombres deben ser descriptivos, independientemente se está nombrando un método, variable, solo para algunas excepciones se usarán abreviaciones cuyo significado se podrá encontrar al final de la sección.

Variables

Para las variables se usará camelCase a excepción de cuando se trata de un atributo de un Record, donde para sus parámetros se usará PascalCase.

Mal uso:

```
n // referring a number
```

```
unr // no way to know what it means
```

Buen uso:

```
int firstNumber;
string clientOrder;
```

El prefijo de un campo dependerá de su modificador de acceso:

- Privado: Se debe poner un guión bajo antes del nombre de la variable
- Estático: Se debe poner una s_ antes del nombre de la variable
- Hilo estático: Se debe poner una t_ antes del nombre del nombre de la variable

Mal uso:

```
int Total = 0;
private string MassiveMessage;
private static Connection connection;
[ThreadStatic]
private static TimeSpan TimeSpan;
```

Buen uso:

```
int accumulator = 0;
private string _message;
private static Queue s_customerQueue;
[ThreadStatic]
private static TimeSpan timeSpan;
```

Constantes

En el caso de las constantes se usará PascalCase, donde cada una de las palabras escritas deberán comenzar con la primera letra en mayúscula.

Mal uso:

```
public const int a = 10;         // Name in lowercase
public const double PI = 3.14159; // Full name in uppercase
```

Buen uso:

```
private const double Pi = 3.1459;
public const int maximumTry = 3;
```

Parámetros

Se deberá usar camelCase junto con nombres que sean descriptivos que reflejan un propósito y sean consistentes, se debe evitar el uso de abreviaciones.

Para los parámetros genéricos (T).

Mal uso:

```
public void Update(ref decimal x, decimal y)
{
    x += y;
}
```

Buen uso:

```
public void UpdateCash(ref decimal clientCash, decimal mount)
{
    clientCash += mount;
}
```

Métodos

Para el nombramiento de métodos se deben llamar según su funcionalidad para facilitar su lectura y comprensión, se usará PascalCase, donde cada una de las palabras escritas deberán comenzar con la primera letra en mayúscula.

Mal uso:

```
public int calculatesum(int a, int b) // Not using PascalCase
{
    return a+b;
}

public void P() // Method name not self-descriptive
{
    Console.WriteLine(“Hello World”);
}
```

Buen uso:

```
public void CalculateSum(int aNumber, int bNumber)
{
    return aNumber + bNumber;
}
```

Interfaces

Para el nombramiento de interfaces se debe empezar con la letra I mayúscula (de Interface) seguido de PascalCase para el nombre de la interfaz. El nombre debe ser descriptivo y representar una capacidad o contrato que implementan las clases.

Cuando sea un comportamiento, se debe usar adjetivos o participios que terminen en “able” o “ible”.

Mal uso:

```
public interface Client // No I prefix
{
      void DoSomething();
}

public interface IL() // Name not self-descriptive
{
      void Login();
}
```

Buen uso:

```
public interface IPrinter
{
      void Print();
}
```

Clases

Para el nombramiento de clases se deberá ocupar PascalCase junto con un sustantivo como nombre que sea descriptivo y claro.

Mal uso:

```
public class class1 // Not using PascalCase
{
    // class parameters and methods
}
```

Buen uso:

```
public class Client
{
    // class parameters and methods
}
```

Métodos get y set

Las propiedades deben ser escritas en PascalCase y deben ser sustantivos que describen el valor.
Se utiliza get para obtener valores y set para asignar respectivamente, siempre se escriben en minúsculas debido a que forman parte del lenguaje.
Se pueden crear propiedades personalizadas para agregar lógica adicional al leer o asignar un valor.

Se utilizan las propiedades personalizadas para:

- *Validar datos* antes de asignarlos.
- *Aplicar reglas de negocio*
- *Transformar información* antes de devolverla.

No se debe ocupar verbos como GetId o SetEdad.

Mal uso:

```
public class Client
{
     public string getName { get; set; }
}
```

Buen uso (propiedades automáticas):

```
public class Client
{
     public string Name { get; set; }
}
```

Buen uso (propiedades personalizadas):

```
public class Client
{
    private int age;
    public int Age
    {
        get{ return age; }
        set
```

```
            {
            if (value < 0)
            {
                throw new ArgumentException(“Age can’t be negative”);
            }
            age = value;
        }
    }
```

Abreviaturas válidas

Abreviatura	**Significado**
DAO	Data Access Object

Prefijos

Especialmente en la capa gráfica se permitirán el uso de prefijos, a continuación se nombrarán los prefijos utilizados y sus significados, no deberán usarse otros que no estén en esta lista.

Prefijo	**Significado**
btn	Button
lbl	Label
txtBk	TextBlock
txtB	TextBox
psswB	PasswordBox
chkB	CheckBox
rdoBtn	RadioButton
cbB	ComboBox
lstB	ListBox
sli	Slider
prgsBar	ProgressBar
img	Image

mdEl	MediaElement
lay	Layout

Mal uso:

```
<TextBox x:Name=”textBoxName” Width=”200” />
<Button x:Name=”login” Content=”Login!” Width=”100” Click=”LoginClick” />
private void LoginClick(object sender, RoutedEventArgs e)
{}
```

Buen uso

```
<TextBlock x:Name=”txtBkTitle” Text=”Super App” />
<Slider x:Name=”sliUserAge” Minimum=”1” Maximum=”120” />
```

Estilos de código

Indentación

Para los espacios de indentación deberán usarse espacios en blanco como medida, y cada nivel debe equivaler a 4 espacios, es decir cada “tab” será igual a cuatro espacios en blanco.

Mal uso:

```
public class Person
{
  private string _name;
  public string Name { get; set; }
}
```

Buen uso:

```
public class Person
{
    private string _name;
    public string Name { get; set; }
    public void PrintName(int repetitions)
    {
        for (int i = 0; i < repetitions; i++)
        {
            Console.WriteLine(name);
        }
    }
}
```

Líneas

Cada línea debe tener por mucho 120 caracteres, en caso de que se exceda esa cantidad

se debe continuar en la siguiente línea cuando se encuentre algunos de los siguientes

caracteres:

- Operadores matemáticos como la suma (+).
- Paréntesis (()), cuando se abren o cierran.
- Comas (,), especialmente en listas de argumentos.
- Llaves ({}), si es el inicio o fin de un bloque de código.
- Corchetes ([]), cuando se declaran o acceden a arreglos.

Mal uso:

```
string myVeryLongText = new string("This is a very long text that
should be split but is not, making it hard to read and maintain in
any code editor without horizontal scrolling.");
```

Buen uso:

```
string textType = new string(“Very long text”);
string myVeryLongText = new string(“This is a “ + textType +
“ I came up with”);
```

Espacios en blanco

Se usará un espacio en blanco cada vez que escribamos una palabra (el nombre de una variable cuenta como una palabra):

Mal uso:

```
int sum=a+b;
Console.WriteLine("Result: "+sum);

for(int i=0;i<10;i++)
{
   Console.WriteLine(i);
}
```

Buen uso:

```
int totalSum = firstNumber + secondNumber;
Console.WriteLine("Calculation result: " + totalSum);

for (int currentIteration = 0; currentIteration < 10; currentIteration++)
{
```

```
        Console.WriteLine("Iteration: " + currentIteration);
    }
```

Uso de llaves

Las llaves de apertura de clases, interfaces y métodos deben estar una línea después de la línea en la que se hizo la declaración. La llave de cierre debe estar en una nueva línea sola e identada para coincidir con el comienzo de la declaración.

Los bloques de estructuras de control (ifs, whiles, for, entre otros) siempre deben estar encerrados con llaves, aunque sólo contengan una línea.

Mal uso:

```
    public class Client(){ // Opening brace in the same line
        public void Greet(){ Console.WriteLine(“Hello!”); }
    }
```

Buen uso:

```
    public class Client()
    {
        public void Greet()
        {
            Console.WriteLine(“Hello!”);
        }
    }
```

Estructuras de control

if - else:

Debe haber un espacio entre **if** y la condición, siempre se debe utilizar llaves **{}**, incluso si es una sola línea.

El **else** debe ir en la línea de abajo de la llave de cierre del **if**.

Mal uso:

```
public void CheckAge(int studentAge)
{
    if(studentAge>=18)
        Console.WriteLine(“The student is an adult.”);
    else
        {Console.WriteLine(“The student is underage.”);}
}
```

Buen uso:

```
public void CheckAge(int studentAge)
{
    if (studentAge >= 18)
    {
        Console.WriteLine(“The student is an adult.”);
    }
    else
    {
        Console.WriteLine(“The student is underage.”);
    }
}
```

else if:

La estructura else if tiene que comenzar en la línea debajo de la última llave del bloque if. La estructura else if tiene que ir seguido por un espacio en blanco antes de la condición y la llave de inicio.

Mal uso:

```
public void CheckGrade(int grade)
{
    if (grade >= 8)
    {
        Console.WriteLine(“Excellent!”);
    }
    else
    if (grade >= 6)     // If should be on the same line as the else
    {
        Console.WriteLine(“Good!”);
    }
    else
    {
     Console.WriteLine(“You can do it better!”);
    }
}
```

Buen uso:

```
public void CheckGrade(int grade)
{
    if (grade >= 8)
    {
        Console.WriteLine(“Excellent!”);
    }
    else if (grade >= 6)
    {
        Console.WriteLine(“Good!”);
    }
    else
    {
     Console.WriteLine(“You can do it better!”);
    }
}
```

Switch:

Debe declararse en una nueva línea y debe tener un espacio antes de la condición.

Cada uno de los case debe ser indentados un nivel.

Mal uso:

```
void Menu()
{
switch(opcion)
{
case 1: // Unindented case
    Console.WriteLine("You select the option 1");
    break;
    default:
     Console.WriteLine("Select a valid option");
     break;
}
```

Buen uso:

```
void Menu()
{
    switch(opcion)
    {
        case 1:
            Console.WriteLine("You select the option 1");
            break;
        default:
            Console.WriteLine("Select a valid option");
            break;
    }
}
```

While:

Debe existir un espacio entre la palabra **while** y la condición.
Siempre se deben utilizar llaves **{}**, incluso cuando el bloque contiene únicamente una línea.

Mal uso:

```
while(true) // There is no space between the while and the condition
     Console.WriteLine("Executing…"); // Braces are missing
```

Buen uso:

```
while(numberA >=0)
{
      Console.WriteLine(numberA);
}
```

Do-While:

En la línea siguiente a la palabra do debe ir la llave de inicio, la palabra **while** debe ir en la línea inmediatamente después de la llave de cierre, separado por un espacio de la condición.

Mal uso:

```
string input;
do{ // Opening brace in the same line
    input = Console.ReadLine();
} while(!CheckInput(input)); // There is no space between the while and
the condition
```

Buen uso:

```
int numberA = 0;
do
{
      Console.WriteLine(numberA);
      i++;
} while (numberA <=5);
```

For:

Debe haber un espacio entre la palabra **for** y la condición, la declaración dentro de los paréntesis debe estar separada por **;** junto con un espacio despues de cada uno, la llave de inicio debera ir en la línea siguiente a la declaración del for.

Mal uso:

```
int sum = 0
for(int actualNumber = 0;actualNumber<10;actualNumber++){
      sum += actualNumber
}
Console.WriteLine(“Sum: “ + sum);
```

Buen uso:

```
int evenCounter = 0;
```

```
for (int currentNumber = 1; currentNumber <= 20; currentNumber++)
{
      if(currentNumber % 2 ==0)
      {
            evenCounter++;
      }
}
Console.WriteLine(“Even numbers: “ + evenCounter);
```

Namespaces

Para los namespaces, los cuales son necesarios para crear claridad para el programador para que cuando se usen se sepa perfectamente qué es lo que contiene. Deberán tener la siguiente estructura:

<Compañía>.(<Producto>|Tecnología>) [.<Feature>] [.<Subnamespace>]

Nombramiento

Se deben usar como prefijos para nombrar los namespaces, estos se recomienda que sean las compañías. Se debe evitar usar nombres dependientes de la versión para el segundo nivel (Producto|Tecnología). Así como usar PascalCase y separar componentes del namespace con puntos, se pueden usar nombres en plural cuando sea apropiado. No usar el mismo nombre para un namespace y tipo en ese namespace y por último no usar nombres genéricos como Element, Node, Log y Message.

Mal uso:

```
fei.ProPractices5_0StudentManagement

Calculator.SquareRoot
```

Buen uso:

```
FEI.Eminus.ActivityAssignment

System.Threading.Tasks
```

Comentarios

El uso de comentarios solo será permitido cuando la implementación que se dio es fuera de lo normal o requiere una explicación más avanzada y para anotar banderas de desarrollo.

Comentarios de línea única

Los comentarios de línea única deben estar rodeados de una línea en blanco, se usarán para expresar que alguna funcionalidad debe codificarse (TO-DO), que hay que arreglarlo (FIX) o que hay que probarlo (TEST).

El mensaje debe estar entre dobles diagonales, un espacio y tres asteriscos (// *** mensaje *** //).

Mal uso:

```
// This method has x functionality
public void SuperCalculator()

// This method works this way
public void foo()
```

Buen uso:

```
// *** FIX *** //

public void FibonacciSequence()
{}

// *** TO-DO *** //

public void EulerSequence()
{}
```

Comentarios de bloque

Los comentarios de bloque se pueden ocupar (no son recomendados) únicamente para explicar una funcionalidad extraña de algún método ya probado.

Para comentar en bloque se usarán los comentarios multi línea (/*, */), se abrirá el comentario y se comenzará a escribir el comentario en la línea siguiente y se cerrará el comentario una línea después de que termine el mensaje.

Mal uso

```
/* This method does not work well
*/

/*
I code this method because...
*/
```

Buen uso

```
/*
This method is built that way because...
*/
```

Comentarios de API pública

Conocidos como *Comentarios de Documentación*, se escriben con tres diagonales (///), es recomendable que incluya un summary mediante una etiqueta XML (<summary></summary>) donde se describe un resumen del método y las etiquetas que correspondan, las más comunes son:

- <param name="..."> Explica cada parámetro
- <returns> Describe qué devuelve la función
- <remarks> Agrega notas adicionales (a mayor detalle)
- <example> Muestra un ejemplo de uso

Ejemplo:

```
/// <summary>
/// Represents a basic calculator
/// </summary>

public class Calculator
{
    /// <summary>
    /// Subtraction of two numbers
    /// </summary>
    /// <param name="firstNumber"> The number we are subtracting of
</param>
    /// <param name="secondNumber"> The number we are subtracting to
firstNumber </param>
    /// <returns> The remainder of firstNumber - secondNumber </returns>
    /// <example>
    /// Example of use:
    /// <code>
    /// var calculator = new Calculator();
    /// int result = calculator.Substraction(10, 3);
    /// </code>
    /// </example>

    public int Subtraction(int firstNumber, int secondNumber)
    {
```

```
            return a - b;
        }
    }
```

Try y Catch

Dentro del try, de preferencia, solo deberá estar el código que pueda lanzar una excepción, y la variable que atrapa la excepción deberá llevar el nombre de `ex`.

Las llaves deberán estar abajo de la declaración del try y catch.

Se evitará el uso de atrapar la excepción genérica, se recomienda el uso de excepciones específicas.

Mal uso:

```
try {
    int[] numbers = {1, 2, 3};
    Console.WriteLine(numbers[5]);
} catch (Exception e){
    Console.WriteLine(e.Message);
}
```

Buen uso:

```
try
{
    int[] numbers = {1, 2, 3};
    Console.WriteLine(numbers[5]);
}
catch (IndexOutOfRangeExeption ex)
{
    logger.error("ERROR: NumberArray.PrintNumber - " + ex);
}
```

Excepciones y errores

Para el manejo de excepciones debemos definir cuándo se debe propagar la excepción y cuando capturarla, cómo se les dará seguimiento y cómo se debe informar al usuario.

Nombramiento de Excepciones en el log

Los mensajes del log deberán tener la siguiente estructura:

> [TIPO_ERROR]: [clase.método] - [Mensaje del error]

Mal uso:

```
logger.error(“ERROR at Animal.MakeSound in “ + exception.Message);
logger.error(“Dog.Bark “ + exception)
```

Buen uso:

```
logger.error(“ERROR: Trainer.CatchPokemon - “ + exception.Message);
logger.error(“FATAL: Calculator.Divide - “ + exception.Message);
```

Manejador de log (Bitácora)

Se debe usar log4Net para darle seguimiento a los errores y excepciones, se deberá mostrar el log en consola y en un archivo .log.

En el mensaje del log se debe indicar en qué método y clase se causó la excepción. De la siguiente manera: “Error en Clase.Método”, ya después irá el log.

Ejemplo incorrecto

```
catch (SQLException e) {
    logger.error(“Error reading something”, e);
}
```

Ejemplo correcto

```
catch (SQLException e) {
    logger.error(“Error at taskDAOImplementation.getTask”, e);
}
```

Excepciones que se mandarán al log debido a su importancia debido a que pueden alterar parte o por completo el funcionamiento del juego.

Con la finalidad de facilitar la **depuración, diagnósito, monitoreo y facilitar el mantenimiento** a largo plazo.

Fatales

- OutOfMemoryException
- StackOverflowException
- FileNotFoundException

Errores

- IOException
- NetworkException
- InvalidOperationException

- NullReferenceException
- ArgumentException
- ArgumentNullException
- DatabaseException

Advertencias

- TimeOutException
- ResourceNotFoundException

Manejo de excepciones

Las excepciones se deben propagar cuando los métodos donde puede ocurrir la excepción se llamarán desde otro método de otra clase, y en la clase desde donde se llama es donde se capturarán las excepciones lanzadas. Un ejemplo de esto es un DAO que puede lanzar un SQLException, dentro del DAO se propagan mientras que capturamos las excepciones cuando los mandamos a llamar.

Todos los errores se deben registrar en la bitácora (logger). En caso de que un error cause el cierre total del programa se debe informar al usuario que hubo un error inesperado y que se cerrará la aplicación, en casos como estos es de suma importancia que se registre en la bitácora. Se debe intentar perder la menor cantidad de información posible.

Siempre que ocurra un error se debe informar a los interesados, por ejemplo, si no se pudo escribir alguna actualización solicitada por el usuario se le deberá avisar mediante un mensaje en la pantalla, si hubo algún error de compilación o vulnerabilidad que no afecte directamente a la UX se deberá registrar en el logger e informar al desarrollador.

Pruebas

Las pruebas deberán abarcar al menos los siguientes casos:

1. Caso ideal
2. Entradas límite
3. Entradas inválidas
4. Comportamiento con datos duplicados
5. Efectos colaterales
6. Excepciones
7. Comprobación de retorno

Nombrado de pruebas

El nombre de una prueba consta de tres partes, la primera es el prefijo Test, seguido por el método (o comportamiento) que se probará y la última parte será el camino que se está calificando (exitoso, entradas mínimas, entradas vacías, etc.).

Mal uso:

```
public void RegisterTaskTest()
public void CalculateSalaryWrong()
public void TestConnectToDatabase()
```

Buen uso:

```
public void TestLoginSuccessful()
public void TestDatabaseConnectionWrongPassword()
public void TestPlayersSyncInvalidToken()
```

Seguridad

Desarrollar un sistema seguro es de gran importancia, especialmente si será un sistema con conexión en red, por lo que mencionamos aquí las principales pautas a seguir para poder mantener la información íntegra y privada.

- Las credenciales de los usuarios deberán estar cifradas
- Las credenciales del sistema para ingresar a bases de datos o algún otro sistema deberán guardarse en otro archivo de configuración al que se accederá desde el código. **JAMÁS DEBERÁ HABER CREDENCIALES EN EL CÓDIGO**
- El código deberá estar libre de smells y no deberá ser susceptible a inyecciones, esto se puede revisar con alguna herramienta de revisión de código como Sonarqube
- Las contraseñas deberán tener una longitud mínima de 8 caracteres y al menos un caracter de los siguientes: mayúsculas, minúsculas y número.

Declaración de IA

Se declara el uso de IA para verificar la gramática del estándar, generación de ejemplos y comprensión de algunos temas.

Bibliografía

BillWagner. (n.d.-a). *Identifier names - rules and conventions - C#*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names

BillWagner. (n.d.-b). *.NET Coding Conventions - C#*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Chidera, I. (2023, January 13). *C# Coding Best Practices – Coding Conventions with Examples*. freeCodeCamp.org. https://www.freecodecamp.org/news/coding-best-practices-in-c-sharp/

De Tinchicus, V. T. L. E. (2020, October 27). *C# / Getter and setter*. El Blog De Tinchicus. https://tinchicus.com/2020/10/27/c-getter-and-setter/

BillWagner. (n.d.). *Documentation comments - C# language specification*. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments

KrzysztofCwalina. (n.d.). Names of Namespaces - Framework design Guidelines. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces

BillWagner. (n.d.-b). Instrucciones de control de excepciones: throw y try, catch, finally - C# reference. Microsoft Learn. https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/statements/exception-handling-statements