Convenciones de código de C# (Guía de programación de C#)

Las convenciones de codificación tienen los objetivos siguientes:

- Crean una apariencia coherente en el código, para que los lectores puedan centrarse en el contenido, no en el diseño.
- Permiten a los lectores comprender el código más rápidamente al hacer suposiciones basadas en la experiencia anterior.
- Facilitan la copia, el cambio y el mantenimiento del código.
- Muestran los procedimientos recomendados de C#.

Microsoft usa las instrucciones de este tema para desarrollar ejemplos y documentación.

Convenciones de nomenclatura

- En ejemplos breves que no incluyen directivas using, use calificaciones de espacio de nombres. Si sabe que un espacio de nombres se importa en un proyecto de forma predeterminada, no es necesario completar los nombres de ese espacio de nombres. Los nombres completos pueden partirse después de un punto (.) si son demasiado largos para una sola línea, como se muestra en el ejemplo siguiente.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- No es necesario cambiar los nombres de objetos que se crearon con las herramientas del diseñador de Visual Studio para que se ajusten a otras directrices.

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.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Convenciones de los comentarios

- 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.

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

- No cree bloques con formato de asteriscos alrededor de comentarios.

Convenciones de lenguaje

En las secciones siguientes se describen las prácticas que sigue el equipo C# para preparar las muestras y ejemplos de código.

String (Tipo de datos)

- Use interpolación de cadenas para concatenar cadenas cortas, como se muestra en el código siguiente.

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

- Para anexar cadenas en bucles, especialmente cuando se trabaja con grandes cantidades de texto, utilice un objeto StringBuilder.

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

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 asignación, o cuando el tipo exacto no sea importante.

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- No use var cuando el tipo no sea evidente desde el lado derecho de la asignación.

```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- No confíe en el nombre de variable para especificar el tipo de la variable. Puede no ser correcto.

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite el uso de `var` en lugar de dynamic.
- Use tipos implícitos para determinar el tipo de la variable de bucle en bucles for y foreach.

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

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
```

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

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

Tipo de datos sin signo

- En general, 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`.

Matrices

- Utilice sintaxis concisa para inicializar las matrices en la línea de declaración.

```
// Preferred syntax. Note that you cannot use var here instead of
string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };


// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one
at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegados

- Utilice sintaxis concisa para crear instancias de un tipo de delegado.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

Instrucciones try-catch y using en el control de excepciones

- Use una instrucción try-catch en la mayoría de casos de control de excepciones.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        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`.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}


// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

Operadores && y ||

- Para evitar excepciones y aumentar el rendimiento omitiendo las comparaciones innecesarias, use && en lugar de & y || en lugar de | cuando realice comparaciones, como se muestra en el ejemplo siguiente.

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

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

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

New (Operador)

- Utilice la forma concisa de la creación de instancias de objeto con tipos implícitos, como se muestra en la siguiente declaración.

```
var instance1 = new ExampleClass();
```

La línea anterior es equivalente a la siguiente declaración.

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

- Utilice inicializadores de objeto para simplificar la creación de objetos.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Control de eventos

- Si va a definir un controlador de eventos que no es necesario quitar más tarde, utilice una expresión lambda.

```
public Form2()
{
    // You can use a lambda expression to define an event handler.
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
// Using a lambda expression shortens the following traditional
definition.
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.

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.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.

```
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.

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

- Utilice tipos implícitos en la declaración de variables de consulta y variables de intervalo.

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.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.

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

- 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.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```