Guía de estilo de programación en C#. Esta guía te ayudara a escribir tus programas de C# mas legibles siguiendo un estilo unificado.

GUÍA DE ESTILOS DE C#

Óscar Álvarez Guerras
http://geeks/blogs/oalvarez

Contenido

1. INTRODUCCIÓN

Esta guía tiene como finalidad proporcionar un conjunto de reglas que nos ayuden a escribir programas en C# con "*buen estilo*". Un código escrito con buen estilo, en cualquier lenguaje, es aquel que tiene las siguientes propiedades:

- Está organizado
- Es fácil de leer
- Es fácil de mantener
- Es fácil detectar errores en él
- Es eficiente

Observando estas reglas, se puede concluir que el objetivo último del estilo de programación es, básicamente *la buena comunicación* en el sentido más amplio, siendo el interlocutor de esa comunicación otras personas que lean el código, los usuarios del programa o las herramientas con las que el código se relaciona (editores de texto, compiladores u otras herramientas colaterales como trazadores o depuradores) o incluso, los manuales del propio software desarrollado.
Las recomendaciones y normas expuestas en esta guía, obtenidas a partir de estándares, experiencias personales y necesidades de los sistemas en la comunidad de desarrolladores de C#, no pretenden formar un conjunto inflexible de estándares de codificación, aunque deben seguirse en la medida de lo posible, proponiendo modificaciones o sugerencias en los casos que se considere oportuno.
Inicialmente, esta guía está orientada a programadores C#, pero la mayor parte de sus principios y reglas se usan a menudo en otros lenguajes de programación.

2. GUÍA DE ESTILO C#

2.1. RECOMENDACIONES GENERALES

Los principios de esta sección sirven para aumentar la legibilidad del código y facilitar su mantenimiento. Nos debemos olvidar que todo el software debe mantener una estructura homogénea y claramente diferenciada.

2.1.1. Encapsulación y Ocultación

La encapsulación y ocultación de información ayudan a organizar mejor el código y evitan el acoplamiento entre funciones.
La **encapsulación** permite agrupar elementos afines del programa. Los subprogramas afines se agrupan en ficheros (unidades), y los datos en grupos lógicos (estructuras de datos).
Ocultación de información: Un subprograma *no necesita* saber lo siguiente:

- La fuente de los parámetros que se le pasan como entrada.
- Para qué servirán sus salidas.
- Qué subprogramas se activaron antes que él.
- Qué subprogramas se activarán después que él.
- Cómo están implementados internamente otros subprogramas.

Para conseguir esto se deben seguir las siguientes reglas:

- Declarar las variables y tipos como locales a los subprogramas que los utilizan.
- Si queremos evitar cambios indeseados en parámetros, pasarlos por valor.
- Un procedimiento sólo debe modificar los parámetros pasados en su llamada.

2.1.2. Comentarios

2.1.2.1. Recomendaciones generales

Los comentarios dan información sobre lo que hace el código en el caso que no sea fácil comprenderlo con una lectura rápida. 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.
Los comentarios se pueden escribir en diferentes estilos dependiendo de su longitud y su propósito. En cualquier caso seguiremos las siguientes **reglas generale**s:

- 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**.

2.1.2.2. Documentación XML. Etiquetas recomendadas

En C#, es posible documentar el código de un programa mediante XML (C# es el único lenguaje de programación de Visual Studio .NET con esta característica) comenzando el comentario con ///.
La documentación debe estar en XML bien formado. Si el XML no está bien formado, se generará un error de advertencia y el archivo de documentación incluirá un comentario que mencione el error encontrado.
Etiquetas básicas recomendadas:

- **<summary>**: se utiliza para describir brevemente un tipo o un miembro de tipo. El texto de esta etiqueta es la única fuente de información sobre el tipo en *IntelliSense*.

  <summary>Descripción***</summary>***

- **<remarks>**: se utiliza para agregar información sobre un tipo, de modo que completa la información especificada con <summary>. ***<remarks>***Descripción detallada***</remarks>***

- **<param>**: se debe utilizar en el comentario de una declaración de método para describir uno de los parámetros del método.

  <param name='NombreParámetro***'>***Descripción***</param>***

- **<returns>**: se debe utilizar en el comentario de una declaración de método para describir el valor devuelto.

  <returns>Descripción***</returns>***
- **<paramref>**:proporciona un modo de indicar que una palabra es un parámetro. El archivo XML se puede procesar de manera que aplique formato a este parámetro de algún modo diferente.

  <paramref name="NombreParámetro***"/>***

- **<exception>**: permite especificar las excepciones que se pueden iniciar. Esta etiqueta se aplica a una definición de método.

  <exception cref="member***">***Descripción***</exception>***
- **<value>**: permite describir una propiedad. Al agregar una propiedad a través del asistente para código se creará una etiqueta <summary> para la nueva propiedad. A continuación, se debe agregar manualmente una etiqueta <value> que describa el valor que representa la propiedad.

  <value>Descripción del valor de la propiedad***</value>***

2.1.2.3. 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 iNumPruebas; // 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 (…)
    {
    …
    } // Fin del FOR
    ```

2.1.2.4. Otras recomendaciones

Otras recomendaciones sobre el uso de comentarios son:

- Cuando modifique el código, mantenga siempre actualizados los comentarios circundantes.
- Después de la implementación, quite todos los comentarios temporales o innecesarios para evitar cualquier confusión en la fase de mantenimiento.
- Use los comentarios para explicar el propósito del código. No los use como si fueran traducciones interlineales.
- Realice los comentarios con un estilo uniforme en toda la aplicación.

2.1.3. 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.

2.1.3.1. 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. Es la opción por defecto del entorno Visual Studio .NET.

2.1.3.2. 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: Espaciado de operadores
  fMedia = iSuma / iCuenta;
  ```

- 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
  AbrirBusqueda (iTarjeta, iMovil);
  ```

2.1.3.3. 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).

2.1.4. Uso de las llaves

La forma que tiene C# de agrupar instrucciones en bloques es utilizar 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 (iContador = 0; iContador < iNumIteraciones;
iContador++)
{
…
}
```

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 (iContador == 0)
{
…
}
else {
…
}
Ejemplo: sentencias tras la llave de cierre
do
{
…
} while (iEstado != iResultOk)
```

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

```
Ejemplo: llaves en bloques con sentencias únicas
if (iContador == 0)
{
iResultado = RET_OK;
}
```

2.1.5. Código sin finalizar

Siempre que partes de la funcionalidad no se implementen, queden inacabadas o sea previsible concluirlas en un futuro, es conveniente documentarlo de modo que sea fácilmente localizable.
Con esta finalidad, el entorno de desarrollo de C# mantiene un listado de tareas pendientes que se pueden obtener directamente de los comentarios de código. Así, cuando se deba completar cualquier tipo de código se documentará con „**//TODO:**„ más la descripción de la funcionalidad pendiente, y ésta aparecerá en el editor de tareas cuando se muestren todas o las de comentario.

```
Ejemplo: documentación de código sin finalizar
private void CalcularBalance ()
{
// TODO: Calcular el Balance de la cuenta
}
```

2.1.6. Conversión de tipos

Cuando sea necesario convertir datos de un tipo a otro, es aconsejable que éstas se hagan de modo explícito, aunque muchas de ellas las realice el compilador automáticamente (implícito).

```
Ejemplo: conversión explícita
long lValor = 3000;
int iValor = (int) lValor;
```

2.2. CONVENCIÓN DE NOMBRES

Los identificadores que dan nombre a la aplicación, clases, métodos, variables, tipos, etc. han de colaborar en la autodocumentación del programa, aportando información sobre el cometido que llevan a cabo (*el qué y no el cómo*). 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.

```
Ejemplo: Uso siglas o acrónimos
int GenararHTML (...)
```

- Utilizar prefijos definidos para cada uno de los elementos.

2.2.1. Variables y objetos

La notación húngara es un sistema usado normalmente en la comunidad de desarrolladores para crear nombres cuando se programa en Windows. Consiste en prefijos en minúsculas que se añaden a los nombres de las variables y que indican su tipo. El resto del nombre indica, lo más claramente posible, la finalidad del elemento.

Prefijo	***Tipo y Significado***	***Ejemplos***
b	Booleano (bool)	bool bNombre;
ch	Carácter (**char**)	char chNombre;
s	Cadena de texto (**string**)	string sNombre;
bt	1 byte (**byte**)	BYTE btNombre;
i	Entero con signo (**int**)	int iNombre;
sh	(short)	short shNombre;
l	Entero largo (**long**)	long lNombre;
db	(double)	double dbNombre;
f	(float)	float fNombre;
d	(decimal)	decimal dNombre ;
dt	(DateTime)	DateTime dtNombre;
st	Estructuras (**struct**)	struct stNombre {};

e	Enumeraciones (enum)	enum eNombre {};
o	Objeto o instancia de una clase	Raiz oRaiz;
ds	(DataSet)	DataSet dsNombre;
dt	(DataTable)	DataTable dtNombre;
dr	(DataRow)	DataRow drNombre;
dc	(DataColum)	DataColum dcNombre;
dv	(DataView)	DataView dvNombre;
drd	(DataReader)	DataReader drdNombre;
ht	(Hashtable)	Hashtable htNombre;

Además de estos tipos estándar, existen otros prefijos que usan junto a éstos para expresar condiciones excepcionales.

u	Tipos primitivos sin signo (**unsigned**)	uint uiNombre;
p	Puntero a objeto (**unsafe code**)	int* piNombre;
arr	Arrays	int[] arriNombre;

2.2.2. Formularios e Interfaces

Prefijo	***Tipo y Significado***	***Ejemplos***
I	Nombres de interfaces	interface INombreInterfaz
Form	Formulario	class FNombreFormulario

2.2.3. Controles Windows

Del mismo modo que con las variables y objetos se notarán del siguiente modo:

Prefijo	***Tipo y Significado***	***Ejemplos***
lbl	Label	lblNombre
lnk	LinkLabel	lnkNombre
btn	Button	btnNombre
txt	TextBox	txtNombre
mnm	MainMenu	mnmNombre
chk	CheckBox	chkNombre
rdb	RadioButton	rdbNombre
grp	GroupBox	grpNombre
pct	PictureBox	pctNombre
pnl	Panel	pnlNombre
dg	DataGrid	dgNombre
lsb	ListBox	lsbNombre
cmb	ComboBox	cmbNombre
lsv	ListView	lsvNombre
trv	TreeView	trvNombre

tbc	TabControl	tbcNombre
prg	ProgressBar	prgNombre
rtb	RichTextBox	rtbNombre
img	ImageList	imgNombre
stb	StatusBar	stbNombre

Nótese que los controles se nombran con sus tres letras más significativas en minúsculas.

2.2.4. Excepciones

Para las propiedades y las clases genéricas no se aplican estas reglas:

Tipo y Significado	***Ejemplos***
Propiedades	string Nombre;
Nombres de clases genéricas	class CNombreClase

2.3. GUÍA DE ESTILO

2.3.1. Ficheros

Por norma general un fichero debe contener una **única clase**. Sólo se incluirán más cuando sea estrictamente necesario.
Las **líneas de código** no deben exceder de los **80 caracteres**, incluyendo las tabulaciones y los comentarios en línea. Las líneas largas deben ser separadas en varios renglones, de modo que al imprimir el código sea legible. Cuando esto ocurra la segunda línea y las posteriores se deben indentar según las circunstancias de cada línea.

```
Ejemplo: Cadena de texto larga
MessageBox.Show("Esto es un ejemplo de línea larga que se debe
indentar en la segunda línea");

Ejemplo: Sentencia condicional
if ( ( (iEstado == Estado.A) && (iFase == Fase.F1) ) ||
( (iEstado == Estado.B) && (iFase == Fase.F2) ) )
{
}
```

Otro aspecto importante es la indentación del código. Esto depende de la configuración de los distintos editores que se usen. En la medida de lo posible se debe indentar mediante **tabuladores** de **4 caracteres**, como se define en el apartado de „Sangría".
La estructura genérica de los ficheros será

- Sentencias *using*
- Definición del *namespace*
- Declaración de la clase
  - ✓ Variables o campos
  - ✓ Delegados y eventos
  - ✓ Propiedades
  - ✓ Métodos

La organización interna de los campos y los métodos en función de su ámbito de acceso será:

- Públicos
- Protegidos
- Privados

2.3.2. Sentencias condicionales y de iteración

2.3.2.1. Sentencia IF

a la sección *if…else* van siempre en una línea sangrada. Aunque está permitido, no se recomienda omitir las llaves cuando se englobe una única sentencia.

```
Ejemplo: sin llaves (no recomendado)
      if (iValor1 < iValor2)
            iValor1=0;
      else
            iValor1=1;
Ejemplo: con llaves
      if (iValor1 < iValor2)
      {
            iValor1=0;
      }
      else {
            iValor1=1;
      }
```

Los *if* anidados deben seguir una secuencia de sangrado lógica.

```
Ejemplo: con llaves
      if (iValor1 < iValor2)
      {
            iValor1=0;
      }
      else if (iValor1 < iValor3)
      {
            iValor1=1;
      }
      else {
            iValor=2;
      }
```

2.3.2.2. Sentencia SWITCH

Los valores asociados a la sentencia *switch* irán en línea aparte sangrada. El bloque de sentencias asociado comenzará en otra línea aparte y también sangrada. Todos los bloques de sentencias pertenecientes al *case* comenzarán en la misma columna. Cada bloque case se deberá separar del siguiente con una línea en blanco cuando englobe muchas líneas de código.

```
Ejemplo: Uso de switch
switch (iValor)
{
      case Valor.Uno:
      sTexto = "Valor A";
      …
      break;
case Valor.Dos:
      sTexto = "Valor B";
      …
      break;
default:
      sTexto = "Valor desconocido";
}
```

Cuando la discriminación se realice sobre tipos numéricos será obligatorio encapsular todos los valores posibles en *enumeraciones* que se autocomenten.

```
Ejemplo: Uso de case
```

```
switch (iValor)
{
      case Valor.Uno: /// Nunca case 1: ¡!!!
            sTexto = "Valor A";
            …
            break;
      …
}
```

Es recomendable comentar el caso en que una sentencia *case* caiga dentro de otra, de modo que las dos ejecuten el mismo código

```
Ejemplo: Uso de case con mismo código
switch (iValor)
{
      case Valor.Uno:: /// Valores impares case Valor.Tres:
            sTexto = "Valor impar";
            break;
      …
}
```

Del mismo modo, cuando una sentencia *case* no tenga código asociado por el motivo que sea, se debe comentar este motivo.

```
Ejemplo: Uso de case sin código
switch (iTipo)
{
      case Tipo.Redondo:
            /// Futura ampliación - sin implementar
            break;
      case Tipo.Cuadrado:
            …
            break;
            …
}
```

2.3.2.3. Sentencias de iteración

Las sentencias pertenecientes a estructuras de iteración o repetición (*for, while, do…while* y *foreach*) van siempre en una nueva sangrada y con su código indentado. Del mismo modo que las sentencias condicionales, se debe evitar la omisión de las llaves.

```
Ejemplo: for sin llaves

for (iContador=0;... ;...)
      iSuma += iTipo;
```

```
Ejemplo: for con llaves
for (iContador=0;... ;...)
      {
            iSuma += iTipo;
            …
      }
```

2.3.3. Variables y objetos

Todas las variables deberán tener nombres significativos. Los nombres que formen una variable comenzarán en mayúsculas y no tendrán ningún carácter de separación (notación Pascal). Al nombre de la variable se le debe anteponer siempre y sin excepción alguna su tipo según el convenio de nombres.

```
Ejemplo: variables correctas
      int iValor;
      string sTextoError;
      CCuadrado oCuadrado;

Ejemplo: variables incorrectas
      int Valor;
      string sTextoerror;
      string stextoerror;
```

Las variables deben declararse siempre al principio de las clases o métodos, una por línea, con comentarios si se estima oportuno e inicializándolas según convenga

```
Ejemplo: declaraciones correctas
private float CalcularRaizCuadrada (int iValor)
{
      // Variables auxiliares
      float fResultado;
      string sTexto = "";
      bool bExacto = false;
      CRaiz oRaiz;
      …
}
```

Los nombres de variables booleanas deben ser lo suficientemente claros como para expresar el valor que encierran ante una pregunta.

```
Ejemplo: variables booleanas
private bool bFinalizado;
if (bFinalizado)
{
}
```

En la medida de lo posible se deben evitar las variables públicas y convertirlas en privadas o protegidas con su propiedad de acceso a ellas.

2.3.3.1. Constantes

Todas las cantidades que permanezcan constantes deben nombrarse con *const*. Este tipo de constantes respetarán las normas comunes de las variables.

```
Ejemplo: uso de constantes
public const int iValor = 10;
```

2.3.4. Propiedades

Para la notación de una propiedad se usará el mismo nombre que la variable sobre la que actúa sin la nomenclatura del tipo.

```
Ejemplo: definición de atributos de clases
class CPoligono
{
private int iNumLados;
public int NumLados
      {
      get
      {
            return iNumlados;
      }
      set
      {
            iNumlados = value;
      }
}
}
```

2.3.5. Operadores

Cuando se usen operadores, principalmente cuando tomen dos parámetros, se deberá dejar un espacio antes y otro después del operador.

```
Ejemplo: uso de operadores
iNumPruebas += 5;
```

Es recomendable el uso de paréntesis para eliminar las ambigüedades que puedan surgir por desconocimiento de las prioridades de los operadores.

```
Ejemplo: uso de paréntesis es operadores
iResultado = (iDato1 * iDato2) / iDato3
```

2.3.6. Regiones

La sentencia **#region** permite especificar un bloque de código que se puede expandir o contraer cuando se utiliza la característica de esquematización del editor de código de Visual Studio.
Dentro de un fichero se deben especificar las siguientes regiones:

- **Clase**: Como ya se ha comentado, es conveniente que exista una única clase por fichero. En el caso de que esto no se pueda cumplir hay que definir una región por cada clase.

```
Ejemplo: región a nivel de clase
#region Clase Balance
...
#endregion
```

- **Variables**: Debe englobar todas las variables definidas

```
Ejemplo: región a nivel de variables
#region Variables
...
#endregion
```

- **Delegados y Eventos**: Si existen delegados y/o eventos se englobarán dentro de una misma región.

Ejemplo: región a nivel de delegados y/o eventos

```
#region Delegados y Eventos
...
#endregion
```

- **Propiedades**: Sobre las propiedades que existan.

```
Ejemplo: región a nivel de propiedades
#region Propiedades
...
#endregion
```

- **Constructores**: Si se han definido constructores personalizados

```
Ejemplo: región a nivel de constructores
#region Constructores
...
#endregion
```

- **Métodos**: Se englobarán todos los métodos

```
Ejemplo: región a nivel de métodos
#region Métodos
...
#endregion
```

Si alguno de los elementos no se encuentran definidos, se debe eliminar la región correspondiente.

2.3.7. 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 según el método **Pascal**. Además debe ser lo suficientemente significativo como para describir la acción que realiza.

```
Ejemplo: uso del estándar verbo-sustantivo
public int CalcularEdad ()
private int CalcularNomina ()
```

Hay que evitar usar nombres imprecisos o que permitan interpretaciones subjetivas

```
Ejemplo: nombre impreciso o ambiguo
public int AnalizarEsto ()
```

Si la lista de parámetros no cabe en una única línea, las siguientes deben estar indentadas hasta la posición en que comienza la lista de parámetros en la primera línea

```
Ejemplo:
private void CalcularReducciones (int iTipo, bool bCeuta,
string sTexto)
{
…
}
```

En los métodos que devuelvan valores debe haber una única sentencia *return* al final de la misma, con un valor que se haya ido rellenando en el cuerpo del método. Incluso si no se devuelve ningún valor, es una buena práctica introducir al final un *return* sin datos

```
Ejemplo: uso de return
bool GetTipo ()
{
      int iTipo;
      … iTipo = Tipo.A;
      … iTipo = Tipo.B;
      return iTipo;
}
```

Cuando se realicen sobrecargas de métodos, todos deben llevar a cabo una función similar.

2.3.8. Clases

Los nombres de las clases deben seguir los conceptos básicos de las funciones, anteponiendo al nombre el prefijo correspondiente definido en el convenio de nombres de clases e interfaces

```
Ejemplo: nombres de clases
class RaizCuadrada
{
…
}
```

Como ya se ha comentado, 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.

```
Ejemplo: nombres de ficheros
RaizCuadrada.cs
class RaizCuadrada
{
}
```

2.3.9. Interfaces

Para la creación de interfaces se deben aplicar las mismas normas descritas para las clases, aplicando su prefijo.

```
Ejemplo: definición de interfaz
interface INuevoInterfaz
{
…
}
```

2.4. OTRAS RECOMENDACIONES

Otras recomendaciones de interés son:

- Está terminantemente prohibido el uso de ***goto***, este comando derrumba los principios básicos de los lenguajes estructurados provocando saltos que dificultan la claridad y el mantenimiento del código.

- No se debe abusar de las sentencias de salida ***break***, ***continue*** o ***exit***, documentando su uso cuando se considere oportuno.

- 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.

  "[…] Hay siempre una sutil razón por la que los programadores siempre quieren desechar el código y empezar de nuevo. La razón es que piensan que el viejo código es una ruina. Y aquí está la observación interesante: probablemente están equivocados. La razón por la que piensan que el viejo código es una ruina es por una ley capital y fundamental de la programación: Es más difícil leer código que escribirlo.

- El idioma de generación de los nombres será siempre el mismo, definido *a priori*.

2.5. MEJORA DE LA EFICIENCIA

- Recordar que el código debe ser mantenido.
- Si la eficiencia del programa no es crítica sustituir instrucciones rápidas por instrucciones comprensibles.
- 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.
- Usar los operadores: ++, --, +=, -= , etc...
- Cuando se pasan estructuras grandes a un subprograma, hacerlo por referencia. Esto evita manipular datos sobre la pila y hace la ejecución más rápida.
- Evitar el abuso de las comparaciones con cadenas de texto. Expresiones del tipo *sValor == "abc"* son muy lentas. Es preferible usar el método Equals (*sValor.Equals("abc")*) o las comparaciones por tamaño si se quiere detectar si la cadena está vacía (*sValor.Lenght == 0*).

APÉNDICE A: NOMENCLATURA DE ESPACIOS DE NOMBRES

Es particularmente importante diseñar correctamente los nombres de los espacios de nombres, para evitar el riesgo de hacer uso de un espacio de nombres idéntico al que utiliza otra persona.

Si dos espacios de nombres con el mismo nombre se instalan en el mismo equipo, se producirá un conflicto de espacios. Microsoft recomienda utilizar nombres de espacios que utilicen el esquema su <NombreEmpresa>.<Tecnología>.

APÉNDICE B: 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

Con el fin de homogeneizar los diferentes mensajes de cara a una estandarización de todos ellos, se generará un documento XML en el que se especificarán los mensajes de error que se pueden producir para un tratamiento mas genérico de la información capturada dentro de las excepciones.

APÉNDICE C: COMPATIBILIDAD CON OTROS LENGUAJES .NET

Para hacer que nuestro código sea compatible con otros leguajes .NET se deben tener en cuenta los siguientes puntos.

C.1. USO DE MAYÚSCULAS Y MINÚSCULAS

Se debe tener cuidado con la distinción entre mayúsculas y minúsculas. C# sí las diferencia, por lo que es perfectamente válido que dos nombres se diferencien únicamente porque algunas letras del primer nombre estén en mayúsculas, mientras que las correspondientes del segundo están en minúsculas. Esto cobra importancia cuando los ensamblados son llamados desde otros lenguajes .NET, como Visual Basic, que no distingue entre mayúsculas y minúsculas.
De este modo, aunque no es aconsejable, si utiliza nombres que se diferencian únicamente por el uso de mayúsculas y minúsculas, es importante hacerlo sólo en situaciones en las que ambos nombres no sean visibles fuera de la unidad del ensamblado. De lo contrario, impedirá que un código escrito en VB .NET sea capaz de acceder correctamente a su unidad.

C.2. NOMBRES Y PALABRAS RESERVADAS

Es importante que los nombres no coincidan con palabras reservadas. Si coincide con las C#, con total seguridad obtendrá un error de sintaxis. Sin embargo, dado que existe la posibilidad de que sus clases sean utilizadas desde código escrito en otros lenguajes, es importante no utilizar palabras reservadas de éstos.
En general las palabras reservadas de C++ son muy similares a las palabras reservadas de C#, por lo que una confusión entre ellas es poco probable.
Por otro lado, es más probable tener problemas con VB .NET, que tiene muchas más palabras reservadas que C#, y no podemos confiar en el estilo de nombres Pascal debido a que VB .NET no distingue entre mayúsculas y minúsculas.