Universidad Tecnológica Nacional
Facultad Regional Córdoba

Ingeniería y Calidad de Software

ESTILOS DE CÓDIGO C#

Grupo 10

Integrantes

Berrera Alejo	85380
Manzur Martin	71247
Marro Tomas	82981
Pary Ronald	87265
Vega Candela	87345
Alaminos Pablo	64194

Docentes

Ing. Laura Covaro
Ing. Mickaela Crespo
Ing. Georgina Gonzáles
Cosntanza Garnero

Curso: 4k4
Fecha Entrega: 30/04/24

ÍNDICE

Introducción

Este documento está diseñado para establecer pautas claras y consistentes para la escritura de código en nuestro equipo. La adopción de un estilo de codificación coherente no solo mejora la legibilidad del código, sino que también facilita la colaboración entre los miembros del equipo y promueve buenas prácticas de programación.

Nuestro objetivo al definir estas reglas de estilo es asegurar que nuestro código sea fácilmente comprensible, mantenga una estructura uniforme y sea escalable a medida que nuestro proyecto evoluciona. Al adherirse a estas directrices, no sólo mejoraremos la calidad del código, sino que también facilitaremos su mantenimiento y futuras expansiones.

A continuación se mencionan una serie de reglas y recomendaciones que deben seguirse al escribir código en C#. Estas reglas abarcan aspectos como la nomenclatura de variables, la disposición del código, el manejo de espacios en blanco y otros elementos importantes para mantener un estilo de codificación claro y consistente.

1. Organización de los ficheros

a) Ficheros de código fuente

- Mantener las clases y los ficheros cortos, con no más de **2.000** líneas de código y que estén claramente divididas en estructuras.
- Crear un fichero para cada clase, con el nombre de la clase como nombre del fichero y la extensión correspondiente. Esta regla puede ser ignorada en los casos en que una clase sea muy dependiente de otra, en cuyo caso podría ser definida en el fichero de la clase importante, o incluso como una clase interna de aquélla.

b) Estructura de directorios

- Crear un directorio para cada nombre de espacio. No utilizar puntos en el nombre de los directorios. Esto hará más fácil la asociación entre directorios y espacios de nombres. Ejemplo: usar **MiEmpresa/MiProyecto/CapaDatos** para el espacio de nombres **MiEmpresa.MiProyecto.CapaDatos**, y no **MiEmpresa/MiProyecto.CapaDatos**.

2. Indentación

a) Espacios en blanco

- Utilizar **Tab** para indentar el código. Nunca utilizar espacios.

b) Ajuste de línea

- Cuando una expresión no quepa en una sola línea de código, dividirla de acuerdo a estos principios:
  - Nueva línea después de una coma.
  - Nueva línea después de un operador aritmético.
  - Buscar operadores con la prioridad más alta posible.
  - Alinear la nueva línea con el comienzo de la sección en la que se encuentra el código.

Ejemplos:

- Luego de una coma:

```
llamadaLarga(expr1, expr2,
             expr3, expr4, expr5);
```

- Luego de un operador aritmético:

```
var = a * b / (c - g + f) +
       4 * z;
```

- Evitar esto:

```
var = a * b / (c - g +
       f) + 4 * z;
```

Evitar el último caso, ya que la división ocurre dentro del paréntesis y esto puede dar lugar a confusión.

- Para mantener las líneas alineadas usar **Tab** y complementar con **espacios**. Este es el único caso donde se permite el uso de espacios para indentar.

3. Comentarios

a) Comentarios de bloque

- Los comentarios de bloque deben ser evitados. Para descripciones de clases y sus miembros, utilice los comentarios con *///* para generar documentación. Si en algún caso se deben utilizar, use el siguiente formato:

```
/* Línea 1
 * Línea 2
 * Línea 3
*/
```

Este formato hace que el bloque sea más legible. Igualmente, los comentarios de bloque raramente son útiles. Básicamente, la utilidad que tienen es que permiten comentar temporalmente grandes bloques de código.

b) Comentarios de línea

- Los comentarios de línea se utilizan para explicar línea a línea el código fuente. También se utilizan para comentar líneas de código temporalmente.
- Estos comentarios deben tener el mismo nivel de indentación que el código que describen.
- La longitud de un comentario no debe exceder la del código que explica. Si esto ocurre, probablemente el código es demasiado complejo, lo que implica un riesgo de errores.

c) Comentarios de documentación

- En la plataforma .NET, Microsoft ha introducido un sistema de generación de documentación basado en comentarios XML. Se deben utilizar comentarios de C# con etiquetas XML internas, y seguir las siguientes directivas:
    - Las líneas deben ser precedidas por tres barras para ser aceptadas como comentarios XML.
    - Las etiquetas pueden ser de dos tipos:

- Elementos de documentación: Etiquetas como **<summary>**, **<param>** o **<exception>**. Estas se relacionan con los elementos de la API del programa que se documenta. Estas etiquetas suelen tener atributos como **name** o **cref**, que son revisados por el compilador, por lo que deben tener valores válidos.
- Formato / referencia: Etiquetas como **<code>**, **<list>** o **<para>**. Estas permiten definir el formato que la documentación debe tener, o crear referencias a elementos del código.

- Ejemplo sencillo:

```
/// <summary>
/// Esta clase es la responsable de ...
/// </summary>
```

- Ejemplo de varias líneas:

```
/// <exception cref="EmailException”>
/// Esta excepción es lanzada cuando ocurre
/// un error durante el envío de un e-mail.
/// </exception>
```

4. Declaraciones

a) Declaraciones de variables locales

- Se recomienda realizar sólo una declaración por línea, ya que esto permite añadir un comentario explicativo a dicha declaración. Ejemplo:

```
int nivel; // nivel de indentación
int tamaño; // tamaño de la tabla
```

- Sin embargo, el uso de nombres claros para las variables puede evitar la necesidad de dichos comentarios explicativos. En este caso, sólo se permite definir dos o más variables en la misma línea cuando todas estas son del mismo tipo de datos. Ejemplo:

```
int nivelIndentacion, tamañoTabla;
```

b) Declaraciones de miembros de clases e interfaces

- Cuando se codifican clases e interfaces con C#, se debe seguir las siguientes reglas:
  - No incluir espacios entre el nombre de un método y los paréntesis donde se encuentran los parámetros del método.
  - La llave de apertura debe aparecer en la línea siguiente a la declaración.

- La llave de clausura debe comenzar una línea, alineada verticalmente con su llave de apertura.

- Ejemplo:

```
class MiEjemplo : MiClase, IMiInterface
{
	int miInt;
	public MiEjemplo(int miInt)
	{
		this.miInt = miInt;
	}
	void Incrementar()
	{
		++miInt;
	}
	void MetodoVacio()
	{
	}
}
```

c) Inicializaciones

- Inicializar las variables locales lo antes posible; si se puede, durante la declaración. Ejemplo:

```
string nombre = miObjeto.Nombre8
int valor = fecha.Hours;
```

Nota: Utilizar la sentencia **using** cuando se inicializa un cuadro de diálogo.

Ejemplo:

```
using (OpenFileDialog openFileDialog = new
OpenFileDialog()) {
...
}
```

5. Sentencias

a) Sentencias simples

- Cada línea debe contener sólo una sentencia.

b) Sentencias de retorno

- Una sentencia de retorno no debe utilizar paréntesis para encerrar el valor de retorno.

No usar: `return (n * (n + 1) / 2);`
Usar: `return n * (n + 1) / 2;`

c) Sentencias if, if-else, if else-if else

- Las sentencias **if**, **if-else** e **if-else if-else** deben tener la siguiente apariencia:

```
if (condicion) {
      // acciones
}

if (condicion) {
      // acciones
} else {
      // otras acciones
}

if(condicion) {
      // acciones
} else if (condicion) {
      // otras acciones
} else {
      // otras acciones más
}
```

d) Sentencias for / foreach

- Una sentencia **for** debe tener la siguiente forma:

```
for (int i = 0; i < 5; ++i) {
      // acciones
}
```

  o de una sola línea (considerar el uso de la sentencia **while** en estos casos):

```
for (initializacion; condicion; cambio) ;
```

- Una sentencia **foreach** debe ser así:

```
foreach (int i in ListaEnteros) {
          // acciones
}
```

Nota: Utilizar llaves incluso cuando haya una sola sentencia en el bucle.

e) Sentencias while/do-while

- Una sentencia **while** debe ser escrita de esta forma:

```
while (condicion) {
       // acciones
}
```

- Una sentencia **while vacía** debe ser así:

```
while (condicion) ;
```

- Una sentencia **do while** debe tener la siguiente forma:

```
do {
      // acciones
} while (condicion);
```

f) Sentencias switch

- Una sentencia **switch** debe ser de la siguiente forma:

```
switch (condicion) {
case A:
      // ...
      break;
case B:
      // ...
      break;
default:
      ...
      break;
}
```

g) Sentencias try-catch

- Una sentencia **try catch** debe tener uno de los siguientes formatos:

```
try {
      // …
} catch (Exception) {}
```

```
try {
      // …
} catch (Exception e) {
...
}
```

```
try {
      // …
} catch (Exception e) {
      //…
} finally {
      //...
```

```
}
```

6. Espaciado

a) Líneas en blanco

- Las líneas en blanco mejoran la legibilidad del código. Separan los bloques de código que están relacionados lógicamente.
- Usar dos líneas en blanco entre:
  - Secciones lógicas de un fichero
  - Definiciones de clases o interfaces
- Usar una línea en blanco entre:
  - Métodos
  - Propiedades
  - Sección de variables locales y la primera sentencia de un método
  - Secciones lógicas dentro de un método
- Las líneas en blanco deben ser indentadas como si contuviera una sentencia, lo que hará más fácil la inserción de código en el futuro.

b) Espacios entre términos

- Debe haber un espacio luego de una coma o un punto y coma. Ejemplos:

```
Prueba(a, b, c);
```

No usar:

```
Prueba(a,b,c);
Prueba( a, b, c );
```

- Debe haber un espacio alrededor de los operadores (excepto los unarios, como el de incremento o la negación lógica). Ejemplos:

```
a = b; // no usar a=b;
for (int i = 0; i < 10; ++i)
      // acciones
```

No usar:

```
for (int i=0; i<10; ++i)
      // acciones
for(int i=0;i<10;++i)
      // acciones
```

c) Formato de tabla

- Un bloque lógico de líneas debe tener un formato de tabla. Ejemplo:

```
string nombre   = "Mr. Ed";
int miValor = 5;
Prueba aPrueba  = new Prueba(5, true);
```

- Usar **espacios** para dar el formato de tabla. No utilizar tabulaciones, ya que se puede perder el formato si se cambia la cantidad de espacios por tabulación.

7. Convenios de nombres

a) Mayúsculas / minúsculas

1. Estilo PasCal
Este convenio determina que la primera letra de cada palabra debe ser mayúscula.

Ejemplo: **C**ontador**P**rueba.

2. Estilo caMel
Este convenio determina que la primera letra de cada palabra debe ser mayúscula, exceptuando la primera palabra.

Ejemplo: contador**P**rueba.

3. Mayúsculas
Este convenio determina que toda la palabra va en letras mayúsculas. Sólo se utiliza para nombres que representan abreviaturas de uno o dos caracteres.

Ejemplos: PI, E.

b) Directivas para asignación de nombres

- Un nombre debe describir la función semántica del elemento, es decir, qué hace, o qué valor representa.
- La notación Húngara es un conjunto de prefijos y sufijos predefinidos que se añaden a los nombres de variables para indicar su tipo. Esto no respeta lo que dice el párrafo anterior, por lo que **no** se debe utilizar esta notación para los nombres. La única excepción a dicha regla es el código de la GUI (Interfaz Gráfica de Usuario). A todos los nombres de campos y variables que contengan elementos de la GUI debe añadirse, como sufijo, su tipo sin abreviar.

  Ejemplos:

```
System.Windows.Forms.Label emailLabel;
System.Windows.Forms.TextBox emailTextBox;
```

- El uso de guión bajo está prohibido.

c) Nombres de clases

- Usar el estilo PasCal.
- Utilizar sustantivos o frases en función del sustantivo.
- No añadir prefijos de clase.

d) Nombres de interfaces

- Usar el estilo PasCal.
- Utilizar sustantivos, frases en función de sustantivo o adjetivos que describan comportamiento.
- Añadir el prefijo I, manteniendo la primera letra del nombre en mayúsculas.

e) Nombres de enumeraciones

- Usar el estilo PasCal, tanto para el nombre de la enumeración como para los valores.
- Utilizar nombres en singular para enumeraciones que obligan a escoger sólo un valor. Ejemplo: la enumeración MessageBoxDefaultButton permite determinar cuál de los botones de un cuadro de mensaje es el predeterminado.
- Utilizar nombres en plural para enumeraciones que permiten escoger valores. Ejemplo: La enumeración MessageBoxButtons permite escoger qué botones se incluyen en un cuadro de mensaje.
- No añadir prefijos ni sufijos al nombre del tipo o de los valores.

f) Nombres de campos estáticos, readonly y constantes

- Usar el estilo PasCal.
- Utilizar sustantivos, frases en función de sustantivo o abreviaciones de sustantivos.

g) Nombres de parámetros y campos no constantes

- Usar el estilo caMel.

h) Nombres de variables

- Usar el estilo caMel.
- Utilizar `i, j, k, l, m, n`, etc. para los contadores locales cuando se utilizan para bucles triviales.

i) Nombres de métodos

- Usar el estilo PasCal.
- Utilizar verbos o frases verbales.

j) Nombres de propiedades

- Usar el estilo PasCal.
- Utilizar sustantivos o frases en función del sustantivo.
- Para las propiedades normales, utilizar el mismo nombre que el campo que almacena su valor.

k) Nombres de eventos

- Usar el estilo PasCal.
- Utilizar verbos.
- Utilizar tiempo presente para eventos generados antes de que algo suceda y pasado para eventos generados luego que algo sucedió.
- Utilizar el sufijo **EventHandler** para el delegado que define la firma de los manejadores del evento.
- Llamar a los parámetros del evento **sender** y **e**.
- Utilizar el sufijo **EventArgs** para las clases de argumentos del evento.

**** Resumen ****

Tipo	***Estilo***	***Notas***
Clase/ estructura	PasCal	
Interface	PasCal	Comienza con I
Enumeración nombre	PasCal	
Enumeración valores	PasCal	
Clases de excepción	PasCal	Finaliza con Exception
Campos públicos	PasCal	
Métodos	PasCal	
Nombres de espacios	PasCal	
Propiedades	PasCal	
Campos protegidos/privados	caMel	
Parámetros	caMel	

8. Datos de cadena

a) Interpolación de cadenas

- 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 System.Text.StringBuilder.

  Ejemplo:

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

9. 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[].

  Ejemplo:

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

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

  Ejemplo:

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

10. Operadores

a) Operadores && y ||

- Use && en vez de & y || en vez de | cuando realice comparaciones, como se muestra en el ejemplo siguiente.

```
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 el 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.

b) Operador new

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

```
var firstExample = new ExampleClass();

ExampleClass instance2 = new();
```

- Las declaraciones anteriores son equivalentes a la siguiente declaración.

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

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

```
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.

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

11. 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.

12. 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 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.

  Ejemplo:

```
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.

  Ejemplo:

```
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.

  Ejemplo:

```
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.

  Ejemplo:

```
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.

  Ejemplo:

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

13. Variables locales con asignación implícita de tipos

a) Tipos implícitos

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

  Ejemplo:

```
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.

Ejemplo:

```
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.

```
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.

- Use la escritura implícita de la variable de bucle en bucles for. En el ejemplo siguiente se usan tipos implícitos en una instrucción for.

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
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.

```
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.

14. Prácticas de programación

a) Visibilidad

- No definir campos públicos; hacerlos privados. Para éstos, no añadir la palabra clave **private**, ya que éste es el valor por defecto. Utilizar propiedades para hacer visibles los valores de dichos campos. La excepción a esta regla son los campos estáticos y constantes.

b) No utilizar números 'mágicos'

- No utilizar valores numéricos constantes directamente en el código fuente. Reemplazarlos luego puede ser un proceso muy propenso a errores e improductivo. Utilizar constantes en su lugar.

  Ejemplo:

```
public class Matematicas
{
	public const double PI = 3.1415926583;
}
```

15. Ejemplos de código

a) Posicionamiento de las llaves

```
namespace EjemploDeLlaves
{
	public enum TipoPrueba {
		Probarme,
		Probarte
	}

	public class Prueba
	{
		TipoPrueba tipoPrueba;
		public TipoPrueba Prueba {
			get {
				return tipoPrueba;
			}
			set {
				tipoPrueba = value;
			}
		}

		void HacerAlgo()
		{
			if (prueba == TipoPrueba.Probarme) {
				//...se hacen cosas
```

```
                        } else {
                              //...se hacen otras cosas
                        }
                  }
            }
      }
```

- Las llaves deben comenzar una nueva línea solo después de:
  - Declaraciones de nombres de espacios
  - Declaraciones de clases, interfaces o estructuras.
  - Declaraciones de métodos.

b) Nombres de variables

- En lugar de:

```
      for (int i = 1; i < numero; ++i) {
            cumpleCriterio[i] = true;
      }
      for (int i = 2; i < numero / 2; ++i) {
            int j = i + i;
            while (j <= numero) {
                  cumpleCriterio[j] = false;
                  j += i;
            }
      }
      for (int i = 0; i < numero; ++i) {
            if (cumpleCriterio[i]) {
                  Console.WriteLine(i + " cumple el criterio");
            }
      }
```

utilizar un sistema más inteligente:

```
for (int candidatoPrimo = 1; candidatoPrimo < numero;
     ++candidatoPrimo)
{
      esPrimo[candidatoPrimo] = true;
}
for (int factor = 2; factor < numero / 2; ++factor) {
      int numeroFactorizable = factor + factor;
      while (numeroFactorizable <= numero) {
            esPrimo[numeroFactorizable] = false;
            factorableNumber += factor;
      }
}
for (int candidatoPrimo = 0; candidatoPrimo < numero;
     ++candidatoPrimo)
{
      if (esPrimo[candidatoPrimo]) {
            Console.WriteLine(candidatoPrimo  + " es
```

```
                primo.");
            }
        }
```

- Nota: Variables contadoras o utilizadas para acceder a propiedades indexadas generalmente deben ser llamadas i, j, k, etc. Pero en casos como éste tiene sentido reconsiderar esa regla. En general, cuando las variables de contador o de acceso a propiedades indexadas se reutilizan, es preferible darles un nombre significativo.