Curso de Patrones y Buenas Prácticas .Net

Buenas Prácticas

Plataforma: Visual Studio .Net
Framework: Microsoft .Net Framework

Documento de Referencia y Capacitación
(Este documento está sujeto a cambios)

Fecha de Creación: 03-Jun-2010
Versión: 2.0.0.0
Autor: Julio Cesar Robles Uribe

Tabla de Contenido

Introducción

Bajo el título de buenas prácticas de programación se incluirán muchas sugerencias a lo largo de las lecciones para subrayar las prácticas que ayudarán a escribir programas más claros, comprensibles y fáciles de mantener, probar y depurar. Estas prácticas son sólo guías, pues los desarrolladores de software siempre tiene su forma particular de resolver un problema, la mayoría de las veces, la solución se da solamente por lo que conocemos o por lo que nos parece más fácil de hacer, pero sin duda usted desarrollará su propio estilo de programación.

También subrayaremos varios errores comunes de programación (problemas que prever para evitar caer en estos errores en sus programas), consejos de desempeño (técnicas que le ayudarán a escribir programas más rápidos y de menor consumo de memoria), explicaciones de portabilidad (técnicas que le ayudarán a escribir programas que puedan ejecutarse con pocos o ningún cambio en diferentes tipos de computadora), observaciones sobre ingeniería de software (ideas y conceptos que afectan y mejoran la arquitectura general de un sistema de software y, en particular, en el caso de los sistemas de software grandes), así como indicaciones de prueba y depuración (sugerencias para probar sus programas y ayudarle a aislar y eliminar sus errores).

Buenas Prácticas

La mayoría de las buenas prácticas son soluciones, ya probadas, a un problema o corrección a errores comunes, que permiten tener una solución que garantiza el rendimiento, la independencia o el menor consumo de recursos ante una situación determinada.

Ahora veremos los casos más comunes, que se dan por inexperiencia o por desconocimiento, del ambiente de desarrollo.

Manejo de Cadenas

Cuando utilizamos cadenas realizamos operaciones de adición o concatenación, inserción y reemplazo, para ello, por lo general, realizamos operaciones simples sin tener en cuenta la utilización de los recursos, como tiempo en procesamiento y memoria utilizada.

Concatenación

Ejemplo Errado:

Esta función realiza la concatenación de datos en una misma cadena, sin tener en cuenta que las cadenas son valores inmutables, es decir, no cambian directamente. Para el caso de las cadenas por cada asignación nueva se crea una nueva instancia de una cadena nueva.

```
static void StringConcat()
{
   //Declarar la cadena
   string cadena = string.Empty;

   //Ciclo para validar la utilizacion de cadenas
   for (int i = 0; i < MAX; i++)
   {
      cadena += i.ToString();
   }
}
```

En este ejemplo al realizar la asignación de la cadena se crea una nueva cadena cada vez que se hace la asignación, este caso, ilustra la utilización en ciclos pero el mismo caso se presenta si se realizan concatenaciones consecutivas con la misma variable.

Ejemplo Correcto:

Si en lugar de utilizar la concatenación de cadenas utilizamos la clase StringBuilder la forma de hacerlo sería la siguiente:

```
static void StringBuilderConcat()
{
   //Declaracion de cadena a concatenar
   StringBuilder sb = new StringBuilder();

   //Ciclo para validar la utilizacion del StringBuilder
   for (int i = 0; i < MAX; i++)
   {
      sb.Append(i.ToString());
   }
}
```

En este caso la utilización del **StringBuilder** permite utilizar una referencia en memoria de la cadena sin crear una nueva instancia lo que mejora el tiempo de procesamiento y la cantidad de memoria utilizada, independientemente si se utiliza en un ciclo o de manera consecutiva.

Remover

Ejemplo Errado:

Al remover datos de una cadena, se crea una nueva cadena, es decir, no se remueven los valores directamente sobre la cadena, sino que se crea una nueva cadena con los valores removidos.

```
static void StringRemove(string strData)
{
   // Longitud de la cadena
   int len = strData.Length - 1;

   //ciclo para reemplazar los datos
   for (int i = 0; i < len; i++)
   {
      strData = strData.Remove(1, 1);
   }
}
```

En este caso, al reemplazar los números del 0 al 9, por cada número reemplazado se crea una nueva cadena.

Ejemplo Correcto:

```
static void StringBuilderRemove(StringBuilder sb)
{
   // Longitud de la cadena
   int len = sb.Length - 1;

   //ciclo para reemplazar los datos
   for (int i = 0; i < len; i++)
   {
      sb.Remove(1, 1);
   }
}
```

La utilización de StringBuilder libera la memoria utilizada y mejora el rendimiento en tiempo del procesamiento de las instrucciones.

Para validar este ejercicio puede descargar el archivo **StringBuilders.zip**, en donde se muestran las funciones de este ejercicio.

Recomendación

- Utilice **StringBuilder** para manejar cadenas en operaciones de concatenación o de remoción de caracteres.
- Sin embargo para operaciones de Reemplazo la mejor opción es utilizar el método **Replace** de la clase **String**.

Para mayor información consulte el siguiente link:

http://www.codeproject.com/KB/cs/StringBuilder_vs_String.aspx

Genéricos

Los genéricos tienen sus orígenes en las plantillas o **Templates** de C++, en donde se podían definir clases que heredaban de otra clase genérica (<T>) que era desconocida.

```
public class Generic<T>
{
    public T Field;
}
```

Los tipos genéricos son una nueva característica en .Net desde la versión 2.0 del lenguaje C# y Common Language Runtime (CLR). Estos tipos agregan el concepto de parámetros de tipo a .NET Framework, lo cual permite diseñar clases y métodos que aplazan la especificación de uno o más tipos hasta que el código de cliente declara y crea una instancia de la clase o del método. Por ejemplo, mediante la utilización de un parámetro de tipo genérico **T**, se puede escribir una clase única que otro código de cliente puede utilizar sin generar el costo o el riesgo de conversiones en tiempo de ejecución u operaciones de conversión **boxing**, como se muestra a continuación:

```
// Declarar Clase Generica
public class GenericList<T>
{
   public T Field;
}

class TestGenericList
{
   private void Main()
   {
      // Declarar una lista de tipo enteros
      GenericList<int> list1 = new GenericList<int>();
      // Declarar una lista de tipos cadena
      GenericList<string> list2 = new GenericList<string>();
   }
}
```

En este caso se crea la clase **GenericList** para manejar una lista de ítems que aun no se sabe de qué tipo será, pero para poder definirla usamos el identificador **T** entre paréntesis angulares (<T>) de esta forma se puede definir la clase, que luego puede tomar el tipo definido en el momento que se utilice.

También se pueden tener métodos con parámetros genéricos que luego toman su valor en el momento de su utilización así:

```
void Swap<T>( ref T left, ref T right)
{
   T temp;
   temp = left;
   left = right;
   right = temp;
}
```

Luego puede utilizar dicho método así:

```
int a = 2;
int b = 3;
Swap<int>(a,b);
```

De igual forma pude omitir el parámetro del tipo <T> ya que el compilador lo completa al momento de generar el componente.

```
Swap(a,b);
```

Uso de los Genéricos

Los tipos genéricos proporcionan la solución a una limitación de las versiones anteriores de Common Language Runtime y del lenguaje C#, en los que se realiza una generalización mediante la conversión de tipos a y desde el tipo base universal **Object**. Con la creación de una clase genérica, se puede crear una colección que garantiza la seguridad de tipos en tiempo de compilación.

Las limitaciones del uso de clases de colección no genéricas se pueden demostrar escribiendo un breve programa que utiliza la clase de colección **ArrayList** de la biblioteca de clases base de .NET Framework. **ArrayList** es una clase de colección muy conveniente, que se puede utilizar sin modificar para almacenar referencias o tipos de valor.

```
// Crear una lista
ArrayList list1 = new ArrayList();
list1.Add(3);
list1.Add(105);

ArrayList list2 = new ArrayList();
list2.Add("Esta lloviendo en Redmond.");
list2.Add("Esta Nevando en las montañas.");
```

Pero esta conveniencia tiene su costo. Cualquier referencia o tipo de valor agregado a un objeto **ArrayList** se convierte implícitamente a **Object**. Si los elementos son tipos de valor, se les debe aplicar la conversión **boxing** cuando se agregan a la lista y la conversión **unboxing** cuando se recuperan. Tanto las operaciones de conversión de tipos como las de conversiones boxing y unboxing degradan el rendimiento; el efecto de las conversiones boxing y unboxing puede ser muy importante en escenarios donde se deben recorrer en iteración colecciones extensas.

La otra limitación es la ausencia de comprobación de tipos en tiempo de compilación; puesto que un objeto **ArrayList** convierte todo a **Object**, en tiempo de compilación no hay forma de evitar que el código de cliente haga lo siguiente:

```
// Crear la lista.
ArrayList list = new ArrayList();

// Adicionar un entero a la lista.
list.Add(3);

// Adicionar una cadena a la lista. Esta operación compila pero causa error luego
list.Add("Esta lloviendo en Redmond.");

int t = 0;

// Esto causa una excepción de tipo invalido
foreach (int x in list)
{
    t += x;
}
```

Aunque es perfectamente válido y a veces intencionado si se crea una colección heterogénea, es probable que la combinación de cadenas y valores **ints** en un objeto **ArrayList** único sea un error de programación, el cual no se detectará hasta el tiempo de ejecución.

En las versiones 1.0 y 1.1 del lenguaje C#, se podían evitar los riesgos de utilizar código generalizado en las clases de colección de la biblioteca de clases base de .NET Framework

escribiendo colecciones propias específicas del tipo. Claro está que, como dicha clase no se puede reutilizar para más de un tipo de datos, se pierden las ventajas de la generalización y se debe volver a escribir la clase para cada uno de los tipos que se almacenarán.

Lo que **ArrayList** y otras clases similares realmente necesitan es un modo de que el código de cliente especifique, por instancias, el tipo de datos particular que se va a utilizar. Eso eliminaría la necesidad de convertir a **T:System.Object** y también haría posible que el compilador realizara la comprobación de tipos. Es decir, **ArrayList** necesita un type parameter. Eso es precisamente lo que los tipos genéricos proporcionan. En la colección genérica **List<T>,** en el espacio de nombres **N:System.Collections.Generic**, la misma operación de agregar elementos a la colección tiene la apariencia siguiente:

```
// Crear una lista de enteros
List<int> list1 = new List<int>();

// Adicionar un valor
list1.Add(3);

// Intentar adicionar una cadena (Esto genera error de compilación)
// list1.Add("Esta lloviendo en Redmond.");
```

En el código de cliente, la única sintaxis que se agrega con **List<T>** en comparación con **ArrayList** es el argumento de tipo en la declaración y creación de instancias. A cambio de esta complejidad de codificación ligeramente mayor, se puede crear una lista que no sólo es más segura que **ArrayList**, sino que también es bastante más rápida, en especial cuando los elementos de lista son tipos de valor.

Ventajas del uso de Genéricos

- Mejoran el rendimiento.
- Código menos propenso a errores.
- Código más legible y fácil de mantener.
- Evitan crear estructuras de datos especializadas.
- Soportan el concepto de sobrecarga, polimorfismo y herencia.
- Permiten especializar el código en función del tipo de dato que se requiere.
- Evitan la utilización de conversiones de tipo (cast) optimizando el rendimiento del código debido a las verificaciones en tiempo de ejecución que se requieren para realizar la conversión.
- Optimizan la manipulación de variables de tipo valor en colecciones, debido a que se requieren menos operaciones de boxing y unboxing.
- Ayudan a generar código menos propenso a errores debido a que las verificaciones de tipo se hacen es en tiempo de compilación en vez de realizarlas en tiempo de ejecución.

Información General de los Genéricos

- Utilice los tipos genéricos para maximizar la reutilización, seguridad de tipos y rendimiento del código.
- El uso más común de genéricos es crear clases de colección.
- La biblioteca de clases de .NET Framework contiene varias nuevas clases de colección genéricas en el espacio de nombres **System.Collections.Generic.** Éstas se deberían utilizar siempre que sea posible en lugar de clases como **ArrayList** en el espacio de nombres **System.Collections**.
- Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.
- Las clases genéricas se pueden restringir para permitir el acceso a métodos en tipos de datos determinados.
- Se puede obtener información sobre los tipos utilizados en un tipo de datos genéricos en tiempo de ejecución y mediante reflexión.

Recomendación

- Utilice genéricos para la definición de listas o tipos especiales que puedan cambiar en el tiempo.

Para mayor información consulte los siguientes links:

http://www.codeproject.com/KB/cs/generics_explained.aspx
http://genericsnet.codeplex.com/

Enumeraciones

Una enumeración o tipo enumerado es un tipo especial de estructura en la que los literales que la componen definen los valores posibles a asignar a una variable

Por ejemplo, una enumeración de nombre Tamaño cuyos objetos pudiesen tomar los valores literales Pequeño, Mediano o Grande se definiría así:

```
enum Tamaño
{
 Pequeño,
 Mediano,
 Grande
}
```

La mayoría de las veces utilizamos valores constantes dentro del código, conocidos como números mágicos, para determinar una condición o realizar alguna validación en particular por ejemplo

```
   //Validar el tamaño
   if (tam == 2)
   {
      //Mostrar la condicion
      Symtem.Console.WriteLine("El tamaño es demasiado grande");
   }
```

Además, estos literales no sólo facilitan la escritura y lectura del código sino que también pueden ser usados por herramientas de documentación, depuradores u otras aplicaciones para sustituir números y mostrar textos muchos más legibles.

```
   //Validar el tamaño
   if (tam == Tamaño.Grande)
   {
      //Mostrar la condicion
      System.Console.WriteLine("El tamaño es demasiado grande");
   }
```

Además, estos literales no sólo facilitan la escritura y lectura del código sino que también pueden ser usados por herramientas de documentación, depuradores u otras aplicaciones para sustituir números y mostrar textos muchos más legibles.

Otra ventaja de usar enumeraciones frente a números mágicos es que éstas participan en el mecanismo de comprobación de tipos de C# y el CLR. Así, si un método espera un objeto Tamaño y se le pasa uno de otro tipo enumerado se producirá, según cuando se detecte la incoherencia, un error en compilación o una excepción en ejecución. Sin embargo, si se hubiesen usado números mágicos del mismo tipo en vez de enumeraciones no se habría detectado nada, pues en ambos casos para el compilador y el CLR serían simples números sin ningún significado especial asociado.

Definición de Enumeraciones

La palabra clave **enum** se utiliza para declarar la enumeración, que consiste en un conjunto de constantes con un nombre determinado. Cada tipo de enumeración tiene un tipo, que puede ser cualquier tipo integral excepto **char**.

```
enum <nombreEnumeración> : <tipoBase>
{
  <literales>
}
```

El tipo predeterminado de los elementos de la enumeración es **int**. De forma predeterminada, el primer enumerador tiene el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1. Por ejemplo si queremos definir una enumeración par los días tendríamos algo así:

```
enum Dias
{
   Lunes,
   Martes,
   Miercoles
   Jueves,
   Viernes,
   Sabado,
   Domingo
}
```

En este caso el Lunes toma el valor 0 y sucesivamente los demás días tomas los valores hasta el Domingo con un valor de 6.

Si quisiéramos cambiar el valor de inicio de enumeración podemos utilizar los la asignación de la siguiente forma:

```
enum Dias
{
   Lunes=1,
   Martes,
   Miercoles
   Jueves,
   Viernes,
   Sabado,
   Domingo
}
```

De esta manera el domingo toma el valor de 7, ya que la enumeración inicia en 1 y los demás valores van incrementando en 1 hasta el último valor de la enumeración.

Uso de las Enumeraciones

Las variables de tipos enumerados se definen como cualquier otra variable (sintaxis <nombreTipo> <nombreVariable>) Por ejemplo:

```
Dias dia;
```

El valor por defecto para un objeto de una enumeración es 0, que puede o no corresponderse con alguno de los literales definidos para ésta.

Si queremos iniciar el valor de la variable enumerada podemos utilizar el nombre de la enumeración y un literal de la misma asó:

```
Dias dia = Dias.Domingo;
```

De igual forma podemos utilizar las enumeraciones en los parámetros de un método, para facilitar su entendimiento y comprensión.

```
public void ValidarDia(Dias d);
```

Recomendación

- Utilice Enumeraciones en lugar de números mágicos.
- Use las enumeraciones cuando necesite agrupar valores constantes bajo un solo nombre.

Para mayor información consulte los siguientes links:
http://msdn.microsoft.com/es-es/library/sbbt4032(v=vs.80).aspx
http://msdn.microsoft.com/es-es/library/4s1w24dx(v=vs.90).aspx
http://www.devjoker.com/contenidos/Tutorial-C/164/Enumeraciones.aspx
http://www.codeproject.com/KB/cs/csenums01.aspx

Manejo de Constantes

Las clases y las estructuras pueden declarar las constantes como miembros. Las constantes son valores que se conocen en tiempo de compilación y no cambian. Las constantes se declaran como campo utilizando la palabra clave **const** antes del tipo del campo. Las constantes se deben inicializar en el momento en que se declaran. Por ejemplo:

```
class Calendario
{
    const int MESES = 12;
    const int SEMANAS = 52;
    const int DIAS = 365;
}
```

La utilización de constantes facilita la legibilidad del código, pero en realidad la constante desaparece al momento de realizar la compilación ya que el compilador reemplaza la palabra que hace referencia a la constante por el valor de la misma, para facilitar el rendimiento de la aplicación.

Para usar la constante simplemente realizamos la referencia al nombre de esta forma:

```
if ( numDias == DIAS)
{
   //Validar si se cumplio el total de dias
   System.Console.WriteLine("se cumplio el total de dias");
}
```

Al momento de realizar la compilación se generaría un cambio en el nombre de la constante de la siguiente forma:

```
if ( numDias == 365)
{
   //Validar si se cumplio el total de dias
   System.Console.WriteLine("se cumplió el total de días");
}
```

Si en realidad queremos independizar el valor de la constante y manejarlas como un conjunto que puede cambiar sin afectar el resto de los demás componentes de la aplicación, la recomendación es crear una o múltiples clases que agrupen los valores de las constantes en un proyecto de tipo librería, declarando los valores de tipo **readonly**.

Por ejemplo definimos un espacio de nombre denominado **Define** para agrupar a las definiciones de enumeraciones y constantes y creamos la clase o clases con los valores constantes a manejar de tipo **readonly** así:

```
namespace MiAplicacion.Define
{
   public static class Constantes
   {
      public static readonly int MESES = 12;
      public static readonly int SEMANAS = 52;
      public static readonly int DIAS = 365;
   }
}
```

De esta forma podemos agrupar las constantes que podrían cambiar de valor sin impactar a las demás clases o componentes que la utilicen, pues la asignación del valor de la constante.

Recomendación

Cuando quiera definir constantes que sean globales a toda la aplicación la mejor opción es crear una librería con una o varias clases con variables de tipo **readonly** que pueden cambiar cuando se necesiten, si se crean constantes, la librerías que ya han sido compiladas ya tienen cambiados

Para mayor información consulte los siguientes links:
http://msdn.microsoft.com/es-es/library/e6w8fe1b(v=vs.80).aspx
http://weblogs.asp.net/psteele/archive/2004/01/27/63416.aspx
http://en.csharp-online.net/const,_static_and_readonly

Tipos Anulables

Un tipo anulable puede representar todos los valores de determinado tipo más la capacidad de poder contener un valor **null**. Es una instancia de la estructura **System.Nullable**. Sólo pueden ser usados con los tipos por valor, ya que los tipos por referencia soportan el valor nulo por defecto.

La razón para usar tipos anulables es para superar el problema de no estar apto para determinar fácilmente si un tipo por valor está vacío o indefinido. Diferente que a los tipos por referencia (que se les puede asignar el valor null) null no es asignable a tipos por valor primitivos.

NOTA: Los tipos **bool** anulables, en una sentencia **if**, **while**, **for**, o alguna otra sentencia lógica o de iteraciones, será equivalente **null** a **false**; no lanzará un error.

Por ejemplo, el siguiente código puede causar un error en tiempo de compilación:

```
int varA = null;
// Error: no se puede convertir en nulo
// 'int' porque es un tipo por valor que no es anulable
```

Usar Tipos anulables como una solución consistente para determinar si un tipo por valor está vacío o indefinido.

Se pueden declarar Tipos anulables de las siguientes dos formas:

```
System.Nullable<T> variable;   // 1era manera

T? variableName;   // 2nda manera
```

Donde T es el tipo del Tipo anulable. T puede ser cualquier tipo por valor incluyendo struct; pero, este no puede ser un tipo por referencia. Por ejemplo:

```
int?    i = 10;
double? d1 = 3.14;
bool?   bandera = null;
char?   letra = 'a';
int?[]  arr = new int?[10];
```

Uso de los Anulables

La necesidad de utilizar este tipo de valores se planteo para interactuar con bases de datos, las cuales, en sus celdas, pueden contener también un valor null.

Si se quiere especificar un valor predeterminado, se puede usar el operador ??. El operador ?? devolverá el valor de la declaración de la izquierda si este no es nulo y si lo es devolverá el valor dado del lado derecho del ??.

```
// Declarar un numero anulable
int? NumeroN = null;
// Evaluar si la variable es nula y si lo es asignar el valor de -1
int Numero = NumeroN ?? -1;   // Numero es -1
// Asignar e la variable anulable un valor
NumeroN = 146;
// Evaluar si la variable es nula y como no lo es retorna el valor de 146
Numero = NumeroN ?? -1;       // Numero es 146
```

Los tipos que aceptan valores NULL tienen las características siguientes:

- Los tipos que aceptan valores NULL representan variables de tipo de valor a las que se puede asignar el valor null. No se puede crear un tipo que acepta valores NULL basado en un tipo de referencia. (Los tipos de referencia ya admiten el valor null.)
- La sintaxis T? es la forma abreviada de System.Nullable < T >, donde T es un tipo de valor. Las dos formas son intercambiables.
- Asigne un valor a un tipo que acepte valores NULL como si se tratara de un tipo de valor normal, por ejemplo, int? x = 10; o double? d = 4.108;
- Utilice la propiedad System.Nullable.GetValueOrDefault para devolver el valor asignado o el valor predeterminado del tipo subyacente si el valor es null, por ejemplo int j = x.GetValueOrDefault();
- Utilice las propiedades de sólo lectura HasValue y Value para comprobar si el valor es NULL y recuperar el valor, por ejemplo if(x.HasValue) j = x.Value;
    - La propiedad HasValue devuelve el valor verdadero si la variable contiene un valor, o falso si es NULL.
    - La propiedad Value devuelve un valor si se ha asignado uno; de lo contrario, devuelve System.InvalidOperationException.
    - El valor predeterminado para una variable de tipo que acepta valores NULL establece HasValue a false. El Value es indefinido.
- Utilice el operador ?? para asignar un valor predeterminado que se aplicará cuando un tipo que acepta valores NULL cuyo valor actual sea NULL se asigne a una tipo que no acepte valore NULL, por ejemplo int? x = null; int y = x ?? -1;
- No se permite la anidación de tipos que aceptan valores NULL. La línea siguiente no se compilará: Nullable<Nullable<int>> n;

Recomendación

Utilice tipos Anulables cuando desee tener una variable que tenga que interactuar con campos de una base de datos o cuando no tenga claro que valor inicial puede tomar la variable, de esta forma puede inicializarla con **null** sin tener que preocuparse por su primer valor.

Para mayor información consulte los siguientes links:
http://msdn.microsoft.com/en-us/library/1t3y8s4s(v=vs.80).aspx
http://msdn.microsoft.com/en-us/library/2cf62fcy(v=vs.80).aspx

Boxing /UnBoxing

Las conversiones boxing y unboxing permiten tratar los tipos de valor como objetos. La aplicación de la conversión boxing a un tipo de valor empaqueta el tipo en una instancia del tipo de referencia **Object**. Esto permite almacenar el tipo de valor en el montón del recolector de elementos no utilizados. La conversión unboxing extrae el tipo de valor del objeto. En este ejemplo, se aplica la conversión boxing a la variable de entero i y ésta se asigna al objeto o.

```
int i = 123;
object o = (object)i;  // boxing
```

A continuación, se puede aplicar la conversión unboxing al objeto o y asignarlo a la variable de entero i:

```
o = 123;
i = (int)o;  // unboxing
```

Las conversiones boxing y unboxing son procesos que consumen muchos recursos. Cuando se aplica la conversión boxing a un tipo de valor, se debe crear un objeto completamente nuevo. Esto puede llevar hasta 20 veces más que una asignación normal. Al aplicar la conversión unboxing, el proceso de conversión puede llevar hasta cuatro veces más que una asignación normal. En menor grado, la conversión de tipos requerida para aplicar la conversión unboxing también es costosa.

Recomendación

Estas operaciones no son recomendadas, pues son demasiado costosas, la recomendación es no utilizar tipos **object**, a no ser que sea completamente necesario, de ser posible utilice genéricos para bajar el acoplamiento.
Las operaciones de cast explicito también realizan internamente boxing/unboxing. Se recomienda utilizar conversiones de tipo segura mediante el operador **AS**.

Para mayor información consulte los siguientes links:
http://msdn.microsoft.com/es-es/library/25z57t8s(v=vs.80).aspx
http://msdn.microsoft.com/es-es/library/yz2be5wk.aspx
http://www.clikear.com/manuales/csharp/c105.aspx
http://www.dijksterhuis.org/exploring-boxing/

Cast : conversión de tipos de forma segura
http://msdn.microsoft.com/en-us/library/cc488006.aspx

Ciclos ForEach/For

La instrucción **foreach** repite un grupo de instrucciones incluidas en el ciclo para cada elemento de una matriz o de un objeto collection. La instrucción foreach se utiliza para recorrer en iteración una colección de elementos y obtener la información deseada, pero no se debe utilizar para cambiar el contenido de la colección, ya que se pueden producir efectos secundarios imprevisibles.

```
class ForEachTest
{
    static void Main(string[] args)
    {
        int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };
        foreach (int i in fibarray)
        {
            System.Console.WriteLine(i);
        }
    }
}
/*
Output:
0
1
2
3
5
8
13
*/
```

Las instrucciones del bucle siguen ejecutándose para cada elemento de la matriz o la colección. Cuando ya se han recorrido todos los elementos de la colección, el control se transfiere a la siguiente instrucción fuera del bloque foreach.

En cualquier punto dentro del bloque foreach, puede salir del bucle utilizando la palabra clave **break** o pasando directamente la iteración siguiente del bucle mediante la palabra clave **continue**.

También se puede salir de un bucle **foreach** mediante las instrucciones **goto**, **return** o **throw**.

Es importante tener en cuenta que con la utilización del ciclo **foreach**, se crea una nueva variable del tipo necesario para recorrer la colección, lo que genera un mayor consumo de recursos, por eso este tipo de ciclos se deben programar con cuidado y no son recomendados con colecciones de muchos datos o cuando los tipos de datos de la colección son complejos.

Es preferible utilizar ciclos **for** sencillos para evitar el alto consumo de memoria, para el ejemplo anterior su implementación equivalente sería de la siguiente forma:

```
int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };

for(int i; i< fibarray.Count; i++)
{
   int dato = fibarray[i];
   System.Console.WriteLine(dato);
}
```

En este caso se utiliza una variable interna dentro del ciclo para hacer referencia al contenido de la colección, de la misma forma como lo realiza el ciclo **foreach**. De igual manera se utiliza la propiedad **Count** de la colección para determinar el límite de la misma.

La forma correcta de implementar este ciclo usando la sentencia for sería la siguiente:

```
int[] fibarray = new int[] { 0, 1, 2, 3, 5, 8, 13 };

// Variable para obtener el contenido de la colección
int dato=0;

// Obtener la cantidad de elementos de la colección o la longitud de la misma
int cant = fibarray.Count;

// Ciclo para recorrer los datos
for(int i; i < cant; i++)
{
   // Obtener el contenido de la colección reusando la misma variable para no
   // consumir memoria adicional sino reutilizar el mismo espacio de memoria
   // varias veces
   dato = fibarray[i];

   // Mostrar el dato
   System.Console.WriteLine(dato);
}
```

Recomendación

Utilice ciclos **foreach**, solamente si la cantidad de datos de la colección es muy pequeña y los tipos de datos son simples, de lo contrario use siempre ciclos **for**, obteniendo la longitud de la colección y reutilizando una variable dentro del ciclo para obtener el valor de la colección.

Para mayor información consulte los siguientes links:
http://www.codeproject.com/KB/cs/foreach.aspx
http://www.david-amador.com/2009/12/csharp-foreach-vs-for-loop/
http://diditwith.net/2006/10/05/PerformanceOfForeachVsListForEach.aspx

DataSets

El **DataSet** es una representación de datos residente en memoria que proporciona un modelo de programación relacional coherente independientemente del origen de datos que contiene. Un **DataSet** representa un conjunto completo de datos, incluyendo las tablas que contienen, ordenan y restringen los datos, así como las relaciones entre las tablas.

Hay varias maneras de trabajar con un DataSet, que se pueden aplicar de forma independiente o conjuntamente. Puede:

- Crear mediante programación una DataTable, DataRelation y una Constraint en un DataSet y rellenar las tablas con datos.
- Llenar el DataSet con tablas de datos de un origen de datos relacional existente mediante DataAdapter.
- Cargar y hacer persistente el contenido de DataSet mediante XML.

El **DataSet** es idóneo para cuando necesitamos un modelo desconectado de la base de datos, pues podemos hacer cambios en el lado del cliente sin que se repliquen inmediatamente en el servidor.

```
// Se Aume que la conexion es valida
string queryString =
  "SELECT CustomerID, CompanyName FROM dbo.Customers";
SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);

DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
```

El DataSet, es un conjunto completo pero a la vez complejo de datos pues al ser replica de la base de datos, algunas veces, se maneja mucha más información de la que en realidad se necesita.

Cuando necesitamos los datos de una sola tabla o vista, no es aconsejable manejar DataSet, en lugar de ello, se aconseja manejar, clases de tipo entidad y como fuente de datos objetos (ObjectDataSource).

Recomendación

Utilice DataSet solamente cuando requiera interactuar con todo el modelo de datos o tener una réplica de la base de datos, en lugar de ello use siempre el modelo de capas y utilice clases de tipo entidad y ObjectDataSource para obtener y acceder a los datos.

Para mayor información consulte los siguientes links:
http://weblogs.asp.net/paolopia/pages/dsvscustomentities.aspx
http://keithelder.net/2007/10/26/datasets-vs-business-entities/
http://cgeers.com/2009/03/14/data-access-objects-with-the-entity-framework/
http://www.c-sharpcorner.com/UploadFile/Mahadesh/9312/

Propagación de Excepciones

Las excepciones son el mecanismo recomendado en la plataforma .NET para propagar los errores que se produzcan durante la ejecución de las aplicaciones (divisiones por cero, lectura de archivos no disponibles, etc.) Básicamente, son objetos derivados de la clase **System.Exception** que se generan cuando en tiempo de ejecución se produce algún error y que contienen información sobre el mismo. Esto es una diferencia respecto a su implementación en el C++ tradicional que les proporciona una cierta homogeneidad, consistencia y sencillez.

Tradicionalmente, el sistema que en otros lenguajes y plataformas se ha venido usando para informar estos errores consistía simplemente en hacer que los métodos en cuya ejecución pudiesen producirse devolvieran códigos que informasen sobre si se han ejecutado correctamente o, en caso contrario, sobre cuál fue el error producido.

Para informar de un error no basta con crear un objeto del tipo de excepción apropiado, sino que también hay pasárselo al mecanismo de propagación de excepciones. A esto se le llama lanzar la excepción, y para hacerlo se usa la siguiente instrucción:

```
throw <objetoExcepciónALanzar>;
```

Una vez lanzada una excepción es posible escribir código que se encargue de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación aborte mostrando un mensaje de error en el que se describe la excepción producida (información de su propiedad **Message**) y dónde se ha producido (información de su propiedad **StackTrace**) Así, dado el siguiente código fuente de ejemplo:

```
try
 <instrucciones>
catch (<excepción1>)
 <tratamiento1>
catch (<excepción2>)
 <tratamiento2>
...
finally
 <instruccionesFinally>
```

El significado de **try** es el siguiente: si durante la ejecución de las **<instrucciones>** se lanza una excepción de tipo **<excepción1>** (o alguna subclase suya) se ejecutan las instrucciones **<tratamiento1>**, si fuese de tipo **<excepción2>** se ejecutaría **<tratamiento2>**, y así hasta que se encuentre una cláusula catch que pueda tratar la excepción producida.

Si no se encontrase ninguna y la instrucción **try** estuviese anidada dentro de otra, se miraría en los **catch** de su **try** padre y se repetiría el proceso. Si al final se recorren todos los **try** padres y no se encuentra ningún catch compatible, entonces se buscaría en el código desde el que se llamó al método que produjo la excepción. Si así se termina llegando al método que inició el hilo donde se produjo la excepción y tampoco allí se encuentra un tratamiento apropiado se aborta dicho hilo; y si ese hilo es el principal (el que contiene el punto de entrada) se aborta el programa y se muestra el mensaje de error con información sobre la excepción lanzada.

Es muy común utilizar un bloque try-catch-finally, como el siguiente:

```
try
{
    Console.WriteLine("En el try");
    Metodo();
    Console.WriteLine("Al final del try");
}
catch (Exception ex)
{
    Console.WriteLine("En el catch capturando la Excepción {0}", ex.Message);
}
finally
{
    Console.WriteLine("finally");
}
```

En este punto se muestra el contenido de la excepción y se maneja según el tipo determinado. Pero en algunos casos cuando se ejecutan métodos dentro de otros y se tienen bloques anidados de try-catch-finally, se realiza la propagación de las excepciones y es común encontrar el siguiente código:

```
try
{
    Metodo();
}
catch (Exception ex)
{
   throw ex;
}
finally
{
    Console.WriteLine("finally");
}
```

Para este caso se cometen dos errores no recomendados, el primero es que se tiene la excepción genérica (**Exception**) envolviendo todos los posibles errores y segundo se crea una instancia de una variable de tipo Exception (ex) y no se realiza ninguna acción con ella.

La forma correcta de manejar esta excepción sería la siguiente:

```
try
{
    <logica>
}
catch(NullReferenceException)
{
   throw ;
}
catch(ArgumentOutOfRangeException)
{
   throw ;
}
catch(InvalidCastException)
{
   throw ;
}
finally
{
    <finally>
}
```

Cosas a evitar al generar excepciones

La siguiente lista identifica prácticas para evitar que al lanzar excepciones:

- Las excepciones no se deben utilizar para cambiar el flujo de un programa como parte de la ejecución ordinaria. Las excepciones sólo debe utilizarse para informar y manejar condiciones de error.

- Las excepciones no se deben devolver como un valor devuelto o parámetro en lugar de ser lanzado.

- No dispare o propague excepciones del tipo **System.Exception**, **System.SystemException**, **System.NullReferenceException** o **System.IndexOutOfRangeException** intencionalmente desde su propio código fuente.

- No crear excepciones que pueden lanzarse en modo de depuración (Debug), pero no en modo de liberación (Release). Para identificar los errores de ejecución durante la fase de desarrollo, utilice Debug.Assert en su lugar.

Recomendación

Se recomienda que las excepciones sean explicitas para poder controlar el flujo según sea el tipo de error y no crear ninguna variable del tipo de la excepción a no ser que se vaya a manipular el error o a realizar alguna acción con la misma.

Las excepciones deben ser utilizadas para comunicar las condiciones excepcionales, No las use para comunicar los eventos que se espera, como llegar a la final de un archivo. Existe un buen numero de excepciones predefinidas en el espacio de nombres del sistema (System) que describe la excepción a una condición que tenga sentido para los usuarios en lugar de definir una nueva clase de excepción y poner la información específica en el mensaje.

Por último, si el código detecta una excepción que no se va a manejar, considere si se debe ajustar esta excepción con información adicional antes de volver a lanzar o crear una clase personalizada para manejar sus excepciones.

Para mayor información consulte los siguientes links:
http://msdn.microsoft.com/en-us/library/ms173160(v=vs.80).aspx
http://msdn.microsoft.com/en-us/library/ms173163.aspx
http://www.blackwasp.co.uk/CSharpThrowingExceptions.aspx
http://www.c-sharpcorner.com/UploadFile/rajeshvs/ExceptionHandlinginCSharp11282005051444AM/ExceptionHandlinginCSharp.aspx

Técnicas de Depuración

En esta sesión se describe cómo utilizar las clases de seguimiento y depuración. Estas clases están disponibles en Microsoft .NET Framework. Puede utilizar estas clases para proporcionar información sobre el rendimiento de una aplicación durante el desarrollo de aplicación o después de la implementación en producción. Estas clases son sólo una parte de las características de instrumentación que están disponibles en .NET Framework.

Ejemplo usando la clase DEBUG

1. Inicie Visual Studio o Visual C# Express.
2. Cree un nuevo proyecto de aplicación de consola de Visual C# denominado **conInfo**.
3. Agregue el siguiente espacio de nombres en parte superior en Class1 o Program.cs. Class1 se crea en Visual Studio. NET. Program.cs se crea en Visual Studio 2005.

```
using System.Diagnostics;
```

4. Para inicializar variables que contienen información acerca de un producto, agregue las siguientes instrucciones de declaración de método **Main** :

```
string sProdName = "Widget";
int iUnitQty = 100;
double dUnitCost = 1.03;
```

5. Especifique el mensaje que genera la clase como primer parámetro de entrada del método **WriteLine**. Presione la combinación de teclas CTRL + ALT + A para asegurarse de que está visible la ventana Resultados.

```
Debug.WriteLine("Debug: Iniciando depuracion de producto");
```

6. Para mejorar la legibilidad, utilice el método sangría para aplicar sangría mensajes posteriores en la ventana resultados:

```
Debug.Indent();
```

7. Para mostrar el contenido de variables seleccionadas, utilice el método WriteLine como sigue:

```
Debug.WriteLine("El Nombre del producto es: " + sProdName);
Debug.WriteLine("El numero de unidades es: " + iUnitQty.ToString());
Debug.WriteLine("El valor por unidad es: " + dUnitCost.ToString());
```

8. También puede utilizar el método **WriteLine** para mostrar el espacio de nombres y el nombre de clase para un objeto existente. Por ejemplo, el código siguiente muestra el espacio de nombres de System.Xml.XmlDocument en la ventana resultados:

```
System.Xml.XmlDocument oxml = new System.Xml.XmlDocument();
Debug.WriteLine(oxml);
```

9. Para organizar el resultado, puede incluir un parámetro de categoría como opcional, segunda entrada del método **WriteLine** . Si especifica una categoría, el formato de la información del mensaje de ventana es "categoría: mensaje." Por ejemplo, la primera línea del código siguiente se muestra "campo: el nombre del producto será Widget" en la salida de ventana:

```
Debug.WriteLine("El Nombre del producto es: " + sProdName,"Categoria:");
Debug.WriteLine("El numero de unidades es: " +
iUnitQty.ToString(),"Categoria:");
Debug.WriteLine("El valor por unidad es: " + dUnitCost.ToString(),
”Categoria:");
Debug.WriteLine ("El Costo total es:" + (iUnitQty * dUnitCost),"Calculo");
```

10. La ventana Resultados puede mostrar mensajes sólo si una condición designada se evalúa como true mediante el método WriteLineIf de la clase Debug . La condición que se va a evaluar es el primer parámetro del método WriteLineIf de entrada. El segundo parámetro de WriteLineIf es el mensaje que sólo aparece si la condición en el primer parámetro se evalúa como true.

```
Debug.WriteLineIf(iUnitQty > 50, "Este mensaje si aparecera");
Debug.WriteLineIf(iUnitQty < 50, "Este mensaje NO aparecera");
```

11. Utilizar el método Assert de la clase Debug para que la ventana Resultados muestra el mensaje sólo si una condición especificada se evalúa como false:

```
Debug.Assert(dUnitCost > 1, "Este mensaje no aparecera");
Debug.Assert(dUnitCost< 1, "Este mensaje aparece si dUnitcost < 1 es falso");
```

12. Crear los objetos TextWriterTraceListener para la ventana de consola (tr1) y un archivo de texto denominado output.txt (tr2) y, a continuación, agregue cada objeto a la colección Listeners depurar :

```
TextWriterTraceListener tr1 = new TextWriterTraceListener(System.Console.Out);
Debug.Listeners.Add(tr1);
TextWriterTraceListener tr2 = new
                          TextWriterTraceListener(System.IO.File.CreateText("Output.txt"))
Debug.Listeners.Add(tr2);
```

13. Para mejorar la legibilidad, utilice el método Quitar sangría para quitar la sangría para los mensajes posteriores que genera la clase Debug . Cuando se utiliza la sangría y los métodos de Quitar sangría juntos, el lector puede distinguir los resultados como grupo.

```
Debug.Unindent();
Debug.WriteLine("Debug: Finaliza la informacion de producto");
```

14. Para asegurarse de que cada objeto de escucha recibe todo su salida, llamar al método Flush para los búferes de clase Debug :

```
Debug.Flush();
```

Descripción de la Técnica

Cuando ejecute un programa, puede utilizar métodos de la clase **Debug** para generar los mensajes que le ayudarán a supervisar la secuencia de ejecución de programa, para detectar errores de funcionamiento o para proporcionar información de la medida de rendimiento. De forma predeterminada, los mensajes que genera la clase **Debug** aparecen en la ventana de resultados de Visual Studio entorno de desarrollo integrado (IDE).

El código de ejemplo utiliza el método **WriteLine** para producir un mensaje seguido de un terminador de línea. Cuando utiliza este método para generar un mensaje, cada mensaje aparece en una línea independiente en la ventana Resultados.

Cuando se utiliza el método **Assert** de la clase **Debug** , la ventana Resultados muestra un mensaje sólo si una condición especificada se evalúa como false. El mensaje también aparece en un cuadro de diálogo modal al usuario. El cuadro de diálogo incluye el mensaje, el nombre del proyecto y el número de instrucción Debug.Assert . El cuadro de diálogo también incluye los botones de tres comandos siguientes:

- **Anular**: La aplicación deja de ejecutarse.
- **Reintentar**: La aplicación entra en modo de depuración.
- **Omitir**: La aplicación continúa.

El usuario debe hacer clic en uno de estos botones antes la aplicación puede continuar.

También puede dirigir resultados de la clase Debug a destinos distinto de la ventana Resultados. La clase Debug tiene una colección denominada Listeners incluye objetos de escucha.

Cada objeto de escucha supervisa la salida de depuración y dirige el resultado a un destino especificado.

Cada escucha en la colección de escucha recibe ningún resultado que genera la clase Debug . Utilice la clase TextWriterTraceListener para definir objetos de escucha . Puede especificar el destino de una clase TextWriterTraceListener a través de su constructor.

Algunos destinos de salida posibles incluyen:

- La ventana de consola mediante la propiedad System.Console.Out .
- Un archivo de texto (.txt) mediante la instrucción System.IO.File.CreateText("FileName.txt") .

Después de crear un objeto TextWriterTraceListener , debe agregar el objeto a la colección Debug.Listeners para recibir resultados.

Mediante la Clase de Seguimiento

También puede utilizar la clase Trace para generar mensajes de ese monitor la ejecución de una aplicación. Las clases Trace y Debug comparten la mayoría de los mismos métodos para generar, incluidos los siguientes:

- WriteLine
- WriteLineIf
- Aplicar sangría
- Quitar sangría
- Assert
- Vaciar

Puede utilizar el seguimiento y las clases **Debug** por separado o conjuntamente en la misma aplicación. En un proyecto de configuración de soluciones de **Debug** y **Trace** y Debug salida están activas. El proyecto genera resultados de ambas de estas clases a todos los objetos de escucha. Sin embargo, un proyecto de configuración de soluciones de versión sólo genera resultados de una clase de seguimiento. El proyecto de configuración de soluciones de versión omite cualquier invocación de método de clase Debug.

```
Trace.WriteLine("Trace: Iniciando Informacion de Producto");
Trace.Indent();

Trace.WriteLine("El nombre del producto es:”+sProdName);
Trace.WriteLine("El nombre del producto es:"+sProdName,"Categoria:" );
Trace.WriteLineIf(iUnitQty > 50, "Este mensaje si aparecera");
Trace.Assert(dUnitCost > 1, "Este mensaje no aparecera");

Trace.Unindent();
Trace.WriteLine("Trace: Finaliza la informacion de producto");

Trace.Flush();

Console.ReadLine();
```

Comprobar que funciona

1. Asegúrese de que Debug es la configuración de la solución actual.
2. Si la ventana Explorador de soluciones no está visible, presione la combinación de teclas CTRL + ALT + L para mostrar esta ventana.
3. Haga clic con el botón secundario del mouse en **conInfo** y, a continuación, haga clic en Propiedades.
4. En el panel izquierdo de la página de propiedad **conInfo**, en la carpeta de configuración, asegúrese de que la flecha apunta a la depuración.
   Nota: En Visual C# Express Edition, haga clic en Depurar en la página **conInfo** .
5. Encima de la carpeta de configuración, en la configuración del cuadro de lista desplegable haga clic en Active (Debug) o depuración y, a continuación, haga clic en Aceptar . En Visual C# professional, haga clic en activo (Depurar) o la depuración en el cuadro de lista desplegable lista de configuración en la página depuración y, a continuación, haga clic en Guardar en el menú archivo.
6. Presione CTRL+ALT+O para mostrar la ventana de resultados.
7. Presione la tecla F5 para ejecutar el código. Cuando aparece el cuadro de diálogo Error de aserción, haga clic en Omitir.
8. En la ventana de consola, presione ENTRAR. Debe finalizar el programa y la ventana de resultados debe muestra el resultado similar al siguiente.

```
Debug: Iniciando depuracion de producto
       El Nombre del producto es: Widget
       El numero de unidades es: 100
       El valor por unidad es: 1.03
       System.Xml.XmlDocument
       Categoria: El Nombre del producto es: Widget
       Categoria: El numero de unidades es: 100
       Categoria: El valor por unidad es: 1.03
       Calculo: El Costo total es: 103
       Este mensaje si aparecera
       ---- DEBUG ASSERTION FAILED ----
       ---- Assert Short Message ----
       Este mensaje aparece si dUnitcost < 1 es falso
       ---- Assert Long Message ----
       at Class1.Main(String[] args)  <%Path%>\class1.cs(34)

       El Nombre del producto es: Widget
       El numero de unidades es: 100
       El valor por unidad es: 1.03
Debug: Finaliza la informacion de producto
Trace: Iniciando Informacion de Producto
El nombre del producto es: Widget
Categoria: El nombre del producto es: Widget
Este mensaje si aparecerá
Trace: Finaliza la informacion de producto
```

9. La ventana de consola y el archivo Output.txt deben mostrar el siguiente resultado:

```
El Nombre del producto es: Widget
El numero de unidades es: 100
El valor por unidad es: 1.03
Debug: Finaliza la informacion de producto
Trace: Iniciando Informacion de Producto
El nombre del producto es: Widget
Categoria: El nombre del producto es: Widget
Este mensaje si aparecerá
Trace: Finaliza la informacion de producto
```

Nota El archivo Output.txt se encuentra en el mismo directorio que el ejecutable conInfo (conInfo.exe). Normalmente, es la carpeta \bin donde se almacena el origen del proyecto

Codigo Completo

```
using System;
using System.Diagnostics;

class Class1
{
   [STAThread]
   static void Main(string[] args)
   {
      string sProdName = "Widget";
      int iUnitQty = 100;
      double  dUnitCost = 1.03;
      Debug.WriteLine("Debug: Iniciando depuracion de producto");
      Debug.Indent();
          Debug.WriteLine("El Nombre del producto es: " + sProdName);
      Debug.WriteLine("El numero de unidades es: " + iUnitQty.ToString());
      Debug.WriteLine("El valor por unidad es: " + dUnitCost.ToString());

      System.Xml.XmlDocument oxml = new System.Xml.XmlDocument();
      Debug.WriteLine(oxml);

      Debug.WriteLine("El Nombre del producto es: " + sProdName,"Categoria:");
      Debug.WriteLine("El       numero       de       unidades       es:       "       +
iUnitQty.ToString(),"Categoria:");
      Debug.WriteLine("El   valor   por   unidad   es:   "   +   dUnitCost.ToString(),
”Categoria:");
      Debug.WriteLine ("El Costo total es:" + (iUnitQty * dUnitCost),"Calculo");

      Debug.WriteLineIf(iUnitQty > 50, "Este mensaje si aparecera");
      Debug.WriteLineIf(iUnitQty < 50, "Este mensaje NO aparecera");

      Debug.Assert(dUnitCost > 1, "Este mensaje no aparecera");
      Debug.Assert(dUnitCost< 1, "Este mensaje aparece si dUnitcost < 1 es falso");

       TextWriterTraceListener tr1 = new TextWriterTraceListener(System.Console.Out);
       Debug.Listeners.Add(tr1);
       TextWriterTraceListener tr2 = new
TextWriterTraceListener(System.IO.File.CreateText("Output.txt"))
       Debug.Listeners.Add(tr2);


      Debug.WriteLine("El Nombre del producto es: " + sProdName);
      Debug.WriteLine("El numero de unidades es: " + iUnitQty.ToString());
      Debug.WriteLine("El valor por unidad es: " + dUnitCost.ToString());
      Debug.Unindent();
      Debug.WriteLine("Debug: Finaliza la informacion de producto");
      Debug.Flush();

      Trace.WriteLine("Trace: Iniciando Informacion de Producto");
      Trace.Indent();

      Trace.WriteLine("El nombre del producto es:”+sProdName);
      Trace.WriteLine("El nombre del producto es:"+sProdName,"Categoria:" );
      Trace.WriteLineIf(iUnitQty > 50, "Este mensaje si aparecera");
      Trace.Assert(dUnitCost > 1, "Este mensaje no aparecera");

      Trace.Unindent();
      Trace.WriteLine("Trace: Finaliza la informacion de producto");

      Trace.Flush();

      Console.ReadLine();
   }

}
```

Solucionar Problemas

- Si el tipo de configuración de solución es la versión , se omite el resultado de clase Debug.
- Después de crear una clase TextWriterTraceListener para un destino concreto, TextWriterTraceListener recibe resultados de la traza y las clases de depuración . Esto ocurre independientemente de si utiliza el método Add de la traza o la clase Debug para agregar TextWriterTraceListener a la clase de escuchas .
- Si agrega un objeto de escuchas para el mismo destino en el seguimiento y las clases Debug , se duplica cada línea de salida, independientemente de si Debug o seguimiento genera el resultado.

```
TextWriterTraceListener myWriter = new TextWriterTraceListener(System.Console.Out);
Debug.Listeners.Add(myWriter);

TextWriterTraceListener myCreator = new TextWriterTraceListener(System.Console.Out);
Trace.Listeners.Add(myCreator);
```

Referencias

http://support.microsoft.com/kb/815788