Colecciones en C#

1 Introducción

El lenguaje C# provee de varios contenedores **(**los más básicos son: **Dictionary, List, Stack, Queue)** en *System.Collections* y *System.Collections.Generic*. La diferencia entre ambos *namespaces* estriba en que el primero permite guardar en esos contenedores objetos de cualquier tipo mediante el uso de polimorfismo (el tipo del elemento del contenedor es **Object**, que es la clase padre última de cualquier posible objeto en C#); mienpas tanto, el segundo permite especificar el tipo del elemento.

Las diferencias enpe ambos son varias. El segundo permite realizar comprobaciones de errores extendidas con respecto al uso que se hace del contenedor. También permite un mejor rendimiento, al conocer de antemano el tipo de los objetos contenidos en él.

El primero muestra peor rendimiento que el segundo, pero por opa parte, el hecho de que los elementos sólo tengan que derivar de **Object** hace que no todos los elementos del contenedor tengan que derivar de la misma clase. En cualquier caso, existen versiones genéricas y no genéricas de los contenedores principales.

```
1     using System.Collections;
2
3     class Prueba {
4      public static void Main()
5      {
6            var l = new List();
7
8            l.Add( 5 );
9            l.Add( 9,5 );
10           l.Add( “Desarrollo e Integración” );
11           l.Add(
12                       new Persona(
13                             “Baltasar”, “jbgarcia@uvigo.es”
14                  );
15     }
16    }
```

El código anterior utiliza una lista que puede contener cualquier objeto que se pueda crear en C#. El código siguiente, sin embargo, sólo permite objetos que deriven de la misma clase, indicada entre símbolos < y >.

```
1       using System.Collections;
2       using System.Collections.Generic;
3
4       class Prueba {
5           public static void Main()
6           {
7                 var l = new List<string>();
8
9                 l.Add( “5” );
10                l.Add( “9,5” );
11                l.Add( “Master de Consultoría” );
12          }
13      }
```

Cada uno debe usarse en su propio escenario. El contenedor genérico (**List<>**), cuando todos los elementos derivan de una misma clase común. El contenedor polimórfico (**List**), cuando se necesitan elementos de distintas clases en el mismo.

Contenedor polimórfico	Contenedor genérico
List	List<T>
HashTable	Dictionary<T,U>
Stack	Stack<T>
Queue	Queue<T>

Tabla 1: Equivalencias enpe contenedores básicos genéricos y no genéricos en C#.

2 Vectores primitivos

Los vectores primitivos permiten el uso de colecciones estáticas (en el sentido de que el número de elementos en la colección es fijo), de cualquier tipo (si bien todos sus elementos deben de derivar de la clase declarada):

```
1           int[] v = { 1, 2, 3 };
2           for(int i = 0; i < v.length; ++i) {
3                 System.Console.Write( v[ i ] );
4           }
```

Los vectores son muy interesantes debido a que tienen un altísimo rendimiento. Todos los contenedores, más potentes, ofrecen el método *CopyTo()* para crear un vector primitivo a partir de su contenido.

Una lista (ver más abajo para más detalles), por ejemplo, se puede convertir en un vector de este modo:

```
1           var l = new List<int>();
2           // …
3           int[] v = new int[ l.Count ];
4           l.CopyTo( v, 0 );
```

3 Contenedores básicos

Los contenedores más importantes son **List**, **Dictionary**, **Queue** y **Stack**. El primero se usa para colecciones de elementos sin ningún orden en especial, el segundo permite almacenar pares de elementos de tal forma que el primero del par permite buscar al segundo; el tercero, es una colección en la que se introducen elementos por un extremo y se sacan por el contrario, mientras el último permite introducir y sacar elementos por el mismo extremo de una colección.

Un ejemplo de manejo de una lista podría ser el siguiente:

```
1               var l = new List<int>();
2
3               l.Add( 1 );
4               l.Add( 2 );
5               l.Add( 3 );
6
7               for(int i = 0; i < l.Count; ++i) {
8                   System.Console.Write( l[ i ] );
9               }
10
11              foreach(var x in l) {
12                  System.Console.Write( x );
13              }
```

Un ejemplo de manejo de un diccionario podría ser el siguiente:

```
1               var d = new Dictionary<string, int>();
2
3               d.Add( "Vigo", 297124 );
4               d.Add( "Orense", 108673 );
5               d.Add( "Lugo", 97635 );
6               d.Add( "Santiago", 94824 );
7
8               foreach(KeyValuePair<string, int> x in d) {
9                   System.Console.Write( x.Key, x.Value );
10              }
11
12              foreach(var x in d) {
13                  System.Console.Write( x.Key, x.Value );
14              }
```

```
15
16              System.Console.Write( d[ "Orense" ] );
```

4 Conseguir que una clase se comporte como una colección

En C#, es posible obtener una clase que sea indistinguible de un contenedor de la librería estándar. Esto es interesante cuando se desea añadir funcionalidad a un contenedor, ya que de otra forma, es tan sencillo como:

```
1                   class ListaPersonas : List<Persona> {}
```

Con esta sencilla línea se crea una lista de objetos **Persona** que emplea para su implementación una lista genérica. Pero por supuesto, tal y como se comentaba más arriba, este no será el caso típico. La creación de una lista de personas indistinguible de la lista original de la librería estándar, se muestra a continuación.

```
1     public class ListaPersonas : IEnumerable, IEnumerable<Persona>
2     {
3       protected List<Persona> lstPersonas = new List<Persona>();
4
5       // Enumerador genérico
6       IEnumerator<Persona> IEnumerable<Persona>.GetEnumerator()
7       {
8             foreach(var Persona in lstPersonas) {
9                   yield return Persona;
10            }
11      }
12
13      // Enumerador básico
14      IEnumerator IEnumerable.GetEnumerator()
15      {
16                  foreach(var Persona in lstPersonas) {
17                        yield return Persona;
18                  }
19      }
20
21      // Indizador
22      public Persona this[int i]
23      {
24          get { return lstPersonas[i]; }
25          set { lstPersonas[i] = value; }
26      }
27    }
```

El primer paso consiste, por tanto, en proveer a la nueva colección de unos enumeradores, de tal forma que sea posible aplicar *foreach* a la misma. De la misma forma, para poder acceder a un elemento en concreto, es posible utilizar el indizador [].

```
1     var lp = new ListaPersonas();
2     // ...
3     foreach(var p in lp) {
4       System.Console.Write( p );
5     }
```

Pero de esta manera, la nueva lista sólo puede ser recorrida. Si además se desea que se puedan crear y eliminar elementos, entonces es mejor hacerla implementar **ICollection**, que incorpora a **IEnumerable**. Así, el código quedaría como:

```
1 public class ListaPersonas : ICollection<Persona> {
2       protected List<Persona> lstPersonas = new List<Persona>();
3
4       public void Add(Persona p)
5       {
6             lstPersonas.Add( p );
7       }
8
9       public int Count {
10            get { return lstPersonas.Count; }
11      }
12
13      public bool IsReadOnly {
14            get { return false; }
15      }
16
17      public void Clear() {
18            lstPersonas.Clear();
19      }
20
21      public bool Contains(Persona p)
22      {
23            return lstPersonas.Contains( p );
24      }
25
26      public bool Remove(Persona p)
27      {
28            return lstPersonas.Remove( p );
29      }
30
31      public void CopyTo(Persona[] lstPersonas, int index)
32      {
33            lstPersonas.CopyTo( lstPersonas, index );
34      }
35
```

```
36      IEnumerator<Persona> IEnumerable<Persona>.GetEnumerator()
37      {
38            foreach(var Persona in lstPersonas) {
39                  yield return Persona;
40            }
41      }
42
43      IEnumerator IEnumerable.GetEnumerator()
44      {
45            foreach(var Persona in lstPersonas) {
46                  yield return Persona;
47            }
48      }
49
50      public Persona this[int i]
51      {
52          get { return lstPersonas[i]; }
53          set { lstPersonas[i] = value; }
54      }
55
56      public ReadOnlyCollection<Persona> AsReadOnly()
57      {
58            return lstPersonas.AsReadOnly();
59      }
60
61 }
```

5 Colecciones de sólo lectura

Las colecciones de sólo lectura se distinguen de las anteriores de que sólo pueden leerse, y no modificar sus elementos, por el contrario.

```
1     public ReadOnlyCollection<Persona> AsReadOnly()
2     {
3       return lstPersonas.AsReadOnly();
4     }
```

Así, en el ejemplo anterior se describía este método, que llama al método *AsReadOnly()* que **List** posee. En realidad, es necesario tener este método debido a que, por limitaciones del modelo de objetos que sigue la máquina .NET, que una colección sea de sólo lectura no implica que sus elementos también lo sean. La clase clave en este caso es **ReadOnlyCollection<T>** que reside en *System.Collections.ObjectModel*. Una forma alternativa, por tanto, de implementar este método, sería:

```
1     public ReadOnlyCollection<Persona> AsReadOnly()
2     {
3       var lst = new Track[ lstTracks.Count ];
4       lstTracks.CopyTo( lst, 0 );
5
6       return new ReadOnlyCollection<Track>( lst );
7     }
```

De la misma manera, una forma de asegurarse de que un vector de cadenas va a permanecer como de sólo lectura, es la siguiente:

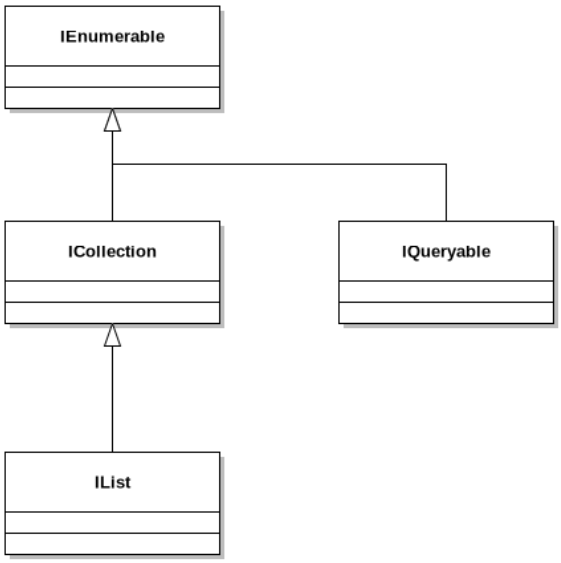
```
1     public static ReadOnlyCollection<string> Headers =
2       new ReadOnlyCollection<string>(
3            new string[]{ "Author", "Album", "Title", "Path" }
4     );
```

Una forma típicamente errónea de tratar de obtener el mismo resultado, es la siguiente:

```
1       public static readonly string[] Headers =
2            new string[]{ "Author", "Album", "Title", "Path" } ;
```

6 Esquema de herencia de las interfaces de colecciones

Las colecciones de C# implementan varias interfaces, que se enumeran a continuación:



1. **IEnumerable<T>**: es la interfaz más básica, que asegura que los elementos pueden recorrerse desde el primero hasta el último. No garantiza métodos para añadir, modificar o borrar, ni siquiera obtener el número de elementos.
2. **ICollection<T>**: la siguiente interfaz en especialización garantiza que se pueden añadir, modificar o borrar elementos, además de obtener el número de ellos.
3. **IList<T>**: el siguiente paso es permitir borrar o insertar en el medio.
4. **IQueryable<T>**: definida en **System.Linq**, garantiza soporte para poder aplicar consultas *Linq* sobre ella.