

INTRODUCCIÓN AL USO DE COLECCIONES.

En muchas aplicaciones se desea poder crear y administrar grupos de objetos relacionados. Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y mediante la creación de **colecciones de objetos**.

Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados.

Las colecciones proporcionan un método más flexible para trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y disminuir dinámicamente en cantidad de elementos, a medida que cambian las necesidades de la aplicación. Para algunas colecciones, puede asignar una clave a cualquier objeto que incluya en la colección para luego recuperarlo rápidamente desde la clave asignada.

Una colección es una clase, de modo que antes de poder agregar elementos a una nueva colección, debe declararla.

Si su colección se limita a elementos de sólo un tipo de datos, puede utilizar una de las clases en el espacio de nombres **System.Collections.Generic**. Una colección genérica cumple con la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando recupera un elemento de una colección genérica, no tiene que determinar su tipo de datos ni convertirlo.

Mencionaremos los tipos de Colecciones en Net (Las más comunes)

Clases de System.Collections.Generic

Clase	Descripción
Collection<T>	Proporciona la clase base para una colección genérica.
Dictionary<TKey, TValue>	Representa una colección de pares de clave y valor que se organizan por claves.
KeyedCollection<TKey, TItem>	Proporciona la clase base abstracta para una colección cuyas claves están incrustadas dentro de los valores.
LinkedList<T>	Representa una lista doblemente vinculada.
LinkedListNode<T>	Representa un nodo en una clase <code>LinkedList<T></code> . Esta clase no puede heredarse.
List<T>	Implementa la interfaz <code>ICollection<T></code> utilizando una matriz cuyo tamaño aumenta dinámicamente cuando es necesario.
Queue<T>	Representa una colección de objetos primero en entrar, primero en salir (FIFO).
SortedDictionary<TKey, TValue>	Representa una colección de pares de clave y valor que se ordenan por claves.
SortedList<TKey, TValue>	Representa una colección de pares de clave y valor que se ordenan por claves según la implementación de la interfaz <code>IComparer<T></code> asociada.
Stack<T>	Representa una colección último en entrar, primero en salir (LIFO) de tamaño variable con instancias del mismo tipo arbitrario.
ReadOnlyCollection<T>	Proporciona la clase base para una colección genérica de sólo lectura.

Nota: Los tipos y métodos genéricos serán vistos más adelante en este curso.

Clases de System.Collections.Specialized

Clase	Descripción
CollectionsUtil	Crea colecciones que omiten el uso de mayúsculas y minúsculas en cadenas.
HybridDictionary	Implementa la interfaz <code>IDictionary</code> utilizando <code>ListDictionary</code> mientras la colección es pequeña; a continuación, cambia a <code>Hashtable</code> cuando la colección aumenta.
ListDictionary	Implementa la interfaz <code>IDictionary</code> utilizando una lista vinculada única. Se recomienda para las colecciones que normalmente contienen 10 elementos o menos.
NameObjectCollectionBase	Proporciona la clase base abstracta para una colección de claves de cadena y valores de objeto asociados a los que se puede tener acceso con la clave o con el índice.
NameValueCollection	Representa una colección de claves de cadena y valores de cadena asociados a los que se puede tener acceso con la clave o con el índice.
OrderedDictionary	Representa una colección de pares de clave y valor que se ordenan por claves o por índices.
StringCollection	Representa una colección de cadenas.
StringDictionary	Implementa una tabla hash con la clave y el valor con establecimiento inflexible de tipos de forma que sean cadenas en lugar de objetos.

Debido a que cada uno de los tipos de Colecciones descritos anteriormente poseen su propio motivo de existencia, no mostraremos ejemplos de estos. En caso que los necesite recomendamos buscar la referencia al tipo de clase necesaria en MSDN.

¿Qué colección utilizar?

Depende del caso y la necesidad de código, más en la mayoría de las implementaciones vera utilizado **List<T>** debido a su simplicidad y efectividad.

List<T> implementa las interfaces **IEnumerable<T>** e **ICollection<T>** con lo cual poseen los métodos necesarios para manejar objetos de manera sencilla.

Bloque Foreach vs. ForNext

La instrucción foreach repite un grupo de instrucciones incrustadas para cada elemento de una matriz o colección de objetos que implementa las interfaces System.Collections.IEnumerable o System.Collections.Generic.IEnumerable<T>. La instrucción **foreach** se utiliza para recorrer la colección iterando en ella y obtener la información deseada, pero no se puede utilizar para agregar o quitar elementos de la colección de origen, ya que se pueden producir efectos secundarios imprevisibles. Si necesita agregar o quitar elementos de la colección de origen, utilice el ciclo **for**. Las instrucciones del ciclo 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 .

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

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

USAR UNA COLECCIÓN SIMPLE

Los ejemplos de esta sección usan la clase genérica **List**, que permite trabajar con una lista de objetos fuertemente tipados.

En el ejemplo siguiente se crea una lista de cadenas y a continuación se itera a través de las cadenas mediante una instrucción **foreach**

```

using System; using
System.Collections.Generic;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            //Crear una lista de Strings
            var artesMarciales = new List<string>();
            artesMarciales.Add("Taekwondo");
            artesMarciales.Add("KungFu");
            artesMarciales.Add("Karate");
            artesMarciales.Add("Aikido");

            // Iterar cada elemento de la lista.
            foreach (var arte in artesMarciales)
            {
                Console.Write(arte + " ");
            }
        }
    }
}

```



Si el contenido de una colección se conoce de antemano, puede utilizar un inicializador de colección para inicializar la colección.

El ejemplo siguiente es igual al ejemplo anterior, excepto que se utiliza un inicializador de colección para agregar elementos a la colección.

```

using System;
using System.Collections.Generic;

```

```

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una lista de Strings usando inicializador de objetos
            var artesMarciales = new List<string> { "Taekwondo", "KungFu", "Karate", "Aikido" };

            // Iterar cada elemento de la lista.
            foreach (var arte in artesMarciales)
            {
                Console.WriteLine(arte + " ");
            }
        }
    }
}

```



Puede utilizar una expresión **for** (C#) en lugar de una instrucción **For Each** para recorrer en iteración una colección. Esto se consigue al obtener acceso a los elementos de la colección mediante el índice de posición. El índice de los elementos comienza en 0 y finaliza en el número de elementos menos 1.

El ejemplo siguiente recorre en iteración los elementos de una colección utilizando **For...Next** en lugar de **For Each**.

```

using System; using
System.Collections.Generic;

```

```

namespace ConsoleApp4
{
    class Program
    {
        static void Main(
string [] args)
        {
            // Crear una lista de Strings usando inicializador de objetos

```

```

var artesMarciales = new List<string> { "Taekwondo", "KungFu", "Karate", "Aikido" };

for (var index = 0; index < artesMarciales.Count; index++)
{
    Console.Write(artesMarciales[index] + " ");
}
}
}
}

```



El ejemplo siguiente elimina un elemento de la colección especificando el objeto que se va a eliminar.

```

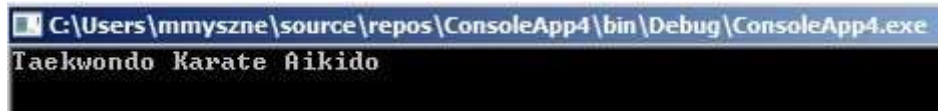
using System;
using System.Collections.Generic;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una lista de Strings usando inicializador de objetos
            var artesMarciales = new List<string> { "Taekwondo", "KungFu", "Karate", "Aikido" };

            // Borrar el elemento de la lista indicando el objeto.
            artesMarciales.Remove("KungFu");

            // Iterar cada elemento de la lista.
            foreach (var arte in artesMarciales)
            {
                Console.Write(arte + " ");
            }
        }
    }
}

```



El ejemplo siguiente elimina elementos de una lista genérica. En lugar de una instrucción **ForEach**, se utiliza una expresión **for** (C#) que itera en orden descendente. Esto se debe a que el método **RemoveAt** hace que los elementos que hay después del elemento eliminado tengan un valor de índice más bajo.

```
using System;
using System.Collections.Generic;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

            // Remover números Pares.
            for (var index = numbers.Count - 1; index >= 0; index--)
            {
                if (numbers[index] % 2 == 1)
                {
                    // Remover el elemento especificando el índice
                    numbers.RemoveAt(index);
                }
            }

            // Iterar la lista
            // Mediante una expresión lambda un Foreach es colocado en el objeto List(T).

            numbers.ForEach(number => Console.Write(number + " "));
        }
    }
}
```



Para el tipo de elementos en **List**, también puede definir su propia clase. En el ejemplo siguiente, la clase **Galaxia** utilizada por List se define en el código.

```
using System;
using System.Collections.Generic;

namespace ConsoleApp4
{
    class Program
    {
        public class Galaxia
        {
            public string Nombre { get; set; }
            public int MegaAniosLuz { get; set; }
        }

        static void Main(string[] args)
        {
            var lasGalaxias = new List<Galaxia>
            {
                new Galaxia() { Nombre="Tadpole", MegaAniosLuz=400},
                new Galaxia() { Nombre="Pinwheel", MegaAniosLuz=25},
                new Galaxia() { Nombre="Milky Way", MegaAniosLuz=0},
                new Galaxia() { Nombre="Andromeda", MegaAniosLuz=3}
            };

            foreach (Galaxia laGalaxia in lasGalaxias)
            {
                Console.WriteLine(laGalaxia.Nombre + " " + laGalaxia.MegaAniosLuz);
            }
            Console.ReadKey();
        }
    }
}
```



```
C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Tadpole 400
Pinwheel 25
Milky Way 0
Andromeda 3
```

Clases de colecciones

Varias colecciones comunes son proporcionadas por .NET Framework. Cada tipo de colección está diseñado para un propósito concreto.

Las enumeraciones como componente de desarrollo

La palabra clave `enum` se utiliza para declarar una **enumeración**, un tipo distinto que consiste en un conjunto de constantes con nombre denominado lista de enumeradores. Normalmente suele ser recomendable definir una enumeración directamente dentro de un espacio de nombres (namespace) para que todas las clases de dicho espacio puedan tener acceso a esta con la misma facilidad. Sin embargo, una enumeración también puede anidarse dentro de una clase o estructura.

De manera predeterminada, el primer valor de la enumeración tiene el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1. Por ejemplo, en la siguiente enumeración, Sab es 0, Dom es 1, Lun es 2, y así sucesivamente.

```
enum Dias { Sab, Dom, Lun, Mar, Mie, Jue, Vie };
```

Los enumeradores pueden usar valores iniciales para invalidar los valores iniciales predeterminados, como se muestra en el ejemplo siguiente.

```
enum Dias { Sab = 1, Dom, Lun, Mar, Mie, Jue, Vie };
```

En esta enumeración, la secuencia de elementos debe iniciarse a partir de 1 en lugar de 0. Sin embargo, se recomienda incluir una constante con el valor 0.

Cada enumeración tiene un tipo subyacente, que puede ser cualquier tipo integral excepto `char`. El tipo subyacente predeterminado de los elementos de la enumeración es `int`. Para declarar una enumeración de otro tipo entero, como `byte`, use el carácter de dos puntos

después del identificador y escriba a continuación el tipo, como se muestra en el ejemplo siguiente.

```
enum Dias : byte { Sab = 1, Dom, Lun, Mar, Mie, Jue, Vie };
```

Los tipos admitidos para una enumeración son **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long** o **ulong**.

A una variable de tipo **Dias** se le puede asignar cualquier valor en el intervalo del tipo subyacente; los valores no se limitan a las constantes con nombre.

Nota: Un enumerador no puede tener espacios en blanco en su nombre

El tipo subyacente especifica la cantidad de almacenamiento asignado a cada enumerador. No obstante, se necesita una conversión explícita para convertir un tipo enum a un tipo entero. Por ejemplo, la siguiente instrucción asigna el enumerador **Dom** a una variable de tipo **int** utilizando una conversión de tipos para convertir de enum a int.

```
int x = (int)Dias.Dom;
```

Programación sólida

Como ocurre con cualquier constante, todas las referencias a los valores individuales de una enumeración se convierten en literales numéricos en tiempo de compilación. Esto puede crear posibles problemas de versiones como se describe en Constantes.

La asignación de valores adicionales a nuevas versiones de enumeraciones o el cambio de los valores de los miembros de enumeración en una nueva versión puede producir problemas para el código fuente dependiente. Los valores **enum** se utilizan a menudo en instrucciones **switch**. Si los elementos adicionales se han agregado al tipo **enum**, la sección predeterminada de la instrucción **switch** se puede seleccionar de forma inesperada.

En el ejemplo siguiente, se declara una enumeración, **Dias**. Dos enumeradores se convierten explícitamente en un número entero y se asignan a variables de número entero.

```
public class EnumTest
{
    enum Dias { Sab, Dom, Lun, Mar, Mie, Jue, Vie };
```

```

static void Main()
{
    int x = (int) Dias.Dom;
    int y = (int) Dias.Vie;
    Console.WriteLine("Dom = {0}", x);
    Console.WriteLine("Vie = {0}", y);
}
}

```



```

C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Dom = 1
Vie = 6

```

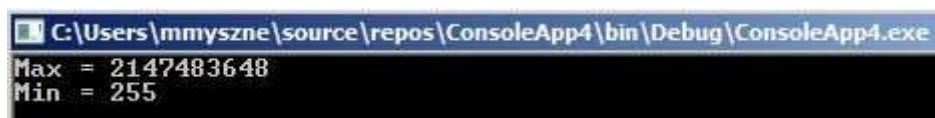
En el ejemplo siguiente, la opción de tipo base se utiliza para declarar un **enum** cuyos miembros son del tipo **long**. Observe que a pesar de que el tipo subyacente de la enumeración es **long**, los miembros de la enumeración todavía deben convertirse explícitamente al tipo **long** mediante una conversión de tipos.

```

public class EnumTest2
{
    enum Rango : long { Max = 2147483648L, Min = 255L };

    static void Main()
    {
        long x = (long)Rango.Max;
        long y = (long)Rango.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}

```



```

C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Max = 2147483648
Min = 255

```

Las Colecciones y Enumeraciones en Propiedades

Si declara una propiedad con el tipo de datos Enum sólo se podrá asignar a la misma los valores de dicha enumeración.