

Programación Web .NET Core

Módulo 2

Uso de colecciones y enumeraciones

Introducción al uso de colecciones

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

- Por la creación de **matrices de objetos**.
- Por 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. En algunas colecciones, **se puede asignar una clave a cualquier objeto** que se incluya, para luego recuperarlo de manera rápida desde la clave asignada.

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



Colecciones genéricas

En el caso de que la colección se limite a elementos de sólo un tipo de datos, se puede usar 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 se recupera un elemento de una colección genérica, no se debe determinar su tipo de datos ni convertirlo.



A continuación, 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 LinkedList<T>. Esta clase no puede heredarse.
List<T>	Implementa la interfaz IList<T> utilizando una matriz cuyo tamaño aumenta dinámicamente cuando es necesario.
Queue <T>	Representa una colección de objetos primero en entrar, primero en salir (FIFO).

Continuación:

Clase	Descripción
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 IComparer<T> asociada.
Stack <T>	Representa una colección última 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 solo lectura.



Arrays multidimensionales (vectores, matrices, cubos, etc.)

Es uno de los tipos de colecciones más simples.
Los **arrays** se instancian **declarando su tipo de dato, dimensiones y cantidad de ítems**.

Una vez declarado un **array**, **la cantidad de ítems no puede ser modificada**.

```
int[] Vector = new int[x];  
int[,] Matriz = new int[x,y];  
int[,,] Cubo = new int[x, y,z];
```

Los **arrays** son de **cualquier tipo de objeto**.

```
Persona[] Personas = new Persona[3];
```



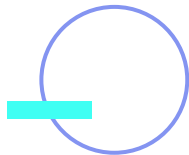
ArrayList

Un **ArrayList** es una colección del tipo lista. A diferencia de los arrays multidimensionales y otras colecciones, **el ArrayList no puede definirse de un tipo de datos específico**, soporta elementos de cualquier tipo de objeto.

Tampoco es necesario declarar la cantidad de elementos, ya que **se redimensionan de manera automática**. Sin embargo, si se declaran con dimensión el rendimiento es mejor ya que no debe redimensionarse dinámicamente cada vez que se agrega un item.

Declaración

```
ArrayList ListaA = new ArrayList();  
ArrayList ListaB = new ArrayList(10);
```



Recorrer sus ítems

```
foreach (object item in ListaA)
{
    //Se muestra el tipo de objeto
    Console.Write("Tipo de objeto: {0}.", item.GetType().FullName);

    //Se decide como mostrar el item segun sea el tipo de objeto
    if (item.GetType() == typeof(System.Int32))
    {
        Console.WriteLine("Valor: {0}", (int) item);
    }
    if (item.GetType() == typeof(string))
    {
        Console.WriteLine("Valor: {0}", item.ToString());
    }
    if (item.GetType() == typeof(Objetos.Persona))
    {
        Persona objPersona = (Persona) item;
        Console.WriteLine("Valor: {0}", objPersona.Nombre + " " + objPersona.Apellido);
    }
}
```

Agregar ítems

```
ListaA.Add(1);  
ListaA.Add("A");  
ListaA.Add(obj);
```

Insertar ítems

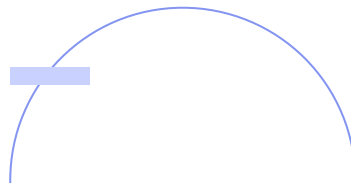
```
ListaA.Insert(1, 2);  
ListaA.Insert(1, "B");  
ListaA.Insert(1, obj);
```

Eliminar ítems

```
ListaA.RemoveAt(0);  
ListaA.Remove("A");
```

Saber si un ítem está contenido en la lista

```
ListaA.Contains("A");
```



Ordenar en forma ascendente

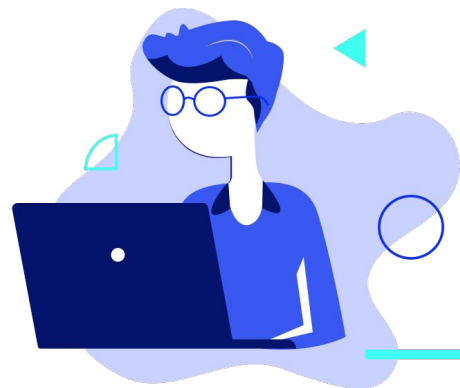
Si el tipo de datos del `ArrayList` no es un tipo primitivo, hay que implementar una interfaz para indicar al método cómo ordenar la colección.

```
ListaC.Sort();
```

Buscar y poder obtener el índice de un elemento en la lista

Para poder utilizar el método `BinarySearch()` la lista debe estar ordenada.

```
ListaC.BinarySearch("Ana")
```



List<T>

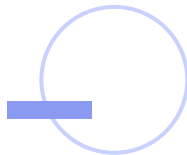
Esta colección es similar a ArrayList. La diferencia es que **las listas deben ser declaradas de un tipo específico**.

Su uso es preferible al de ArrayList, ya que es posible **detectar en tiempo de compilación que no se agreguen por error elementos de un tipo diferente al declarado**.

Son muy eficientes cuando se accede a ellas secuencialmente.

Declaración

```
List<string> ListaA = new List<string>();  
List<string> ListaB = new List<string>(10);
```

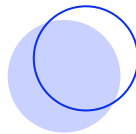


Recorrer sus ítems

Como se conoce el tipo de dato, **no es necesario determinar el tipo de objeto**, como en el caso de un `ArrayList`:

```
foreach (Persona item in ListaC)
{
    Console.WriteLine("Apellido: {0}, {1}",item.Apellido, item.Nombre);
}
```

El resto de las operaciones (agregar, insertar, etc.) se realizan en forma análoga a la clase `ArrayList`.

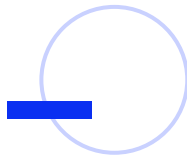


SortedList<k, v>

La principal diferencia con respecto a la lista, es que **se accede a sus elementos a través de una clave**. Asimismo, la lista se ordena independientemente del orden en el que se agregan los ítems.

Declaración

```
SortedList<string, int> ListaOrdenada = new SortedList<string, int>();
```



Agregar ítems

```
ListaOrdenada.Add("C", 300);  
ListaOrdenada.Add("A", 100);  
ListaOrdenada.Add("B", 200);
```

Acceder a un valor través de su clave

```
Console.WriteLine(ListaOrdenada["A"]);  
Console.WriteLine(ListaOrdenada["B"]);  
Console.WriteLine(ListaOrdenada["C"]);
```

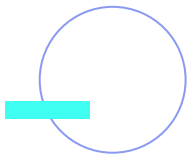
Recorrer sus ítems

```
foreach (KeyValuePair<string, int> item in ListaOrdenada)  
{  
    Console.WriteLine("Clave: {0}. Valor: {1}",item.Key, item.Value);  
}
```



Queue<T>

Esta clase implementa una cola. El primer elemento en entrar, es el primero en salir (*FIFO*). A medida que se accede a sus elementos, estos van siendo eliminados.



Declaración

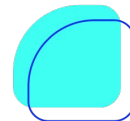
```
Queue<Persona> Cola = new Queue<Persona>();
```

Agregar ítems

```
Cola.Enqueue(new Persona("Ana", "González"));  
Cola.Enqueue(new Persona("Pedro", "Benítez"));  
Cola.Enqueue(new Persona("Isabel", "Pérez"));
```

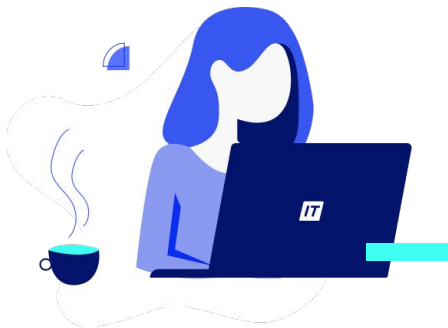

Recorrer y vaciar sus ítems

```
while (Cola.Count > 0)
{
    Persona item = Cola.Dequeue();
    Console.WriteLine("{0}, {1}", item.Apellido, item.Nombre);
}
```



Stack<T>

Esta clase implementa una pila. El último elemento en entrar, es el primero en salir (*LIFO*). A medida que se accede a sus elementos estos van siendo eliminados.



Declaración

```
Stack<Persona> Pila = new Stack<Persona>();
```

Agregar ítems

```
Pila.Push(new Persona("Ana", "González"));  
Pila.Push(new Persona("Pedro", "Benítez"));  
Pila.Push(new Persona("Isabel", "Pérez"));
```

Recorrer y vaciar sus ítems

```
while (Pila.Count > 0)
{
    Persona item = Pila.Pop();
    Console.WriteLine("{0}, {1}", item.Apellido, item.Nombre);
}
```

Nota: Los tipos y métodos genéricos serán vistos más adelante en la carrera .NET.

Clases de System.Collections.Specialized

Clase	Descripción
CollectionsUtil	Crea colecciones que omiten el uso de mayúsculas y minúsculas en cadenas
HybridDictionary	Implementa la interfaz IDictionary usando ListDictionary mientras la colección es pequeña, a continuación, cambia a Hashtable cuando la colección aumenta.
ListDictionary	Implementa la interfaz IDictionary usando una lista vinculada única. Se recomienda para las colecciones que normalmente contienen 10 elementos o menos.
NameObjectCollectionBase	Proporciona la clase 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.

Continuación:

Clase	Descripción
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.



¿Qué colección utilizar?

Depende del caso y la necesidad de código, más **en la mayoría de las implementaciones verá usado `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

La instrucción **foreach** repite un grupo de instrucciones incrustadas para cada elemento de una matriz o colección de objetos, que implementa las interfaces siguientes:

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 usar 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 use 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** usando la palabra clave **break** o **pasar 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 a continuación, se crea una lista de cadenas y luego se itera a través de las cadenas, mediante una instrucción **foreach**.




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

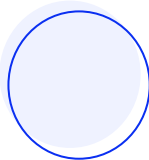

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una lista de Strings
            var artesMarciales = new List<string>();
            artesMarciales.Add("Taekwondo");
            artesMarciales.Add("KungFu");
            artesMarciales.Add("Karate");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

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



```
C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe  
Taekwondo KungFu Karate Aikido
```

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 + " ");
}
}
```



C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Taekwondo KungFu Karate Aikido

Puede utilizar una expresión **for** (C#) en lugar de una instrucción **foreach** 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** en lugar de **foreach**.

```
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 = newList<string>{ "Taekwondo", "KungFu", "Karate", "Aikido" };
        }
    }
}
```

...

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



C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Taekwondo KungFu Karate Aikido

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

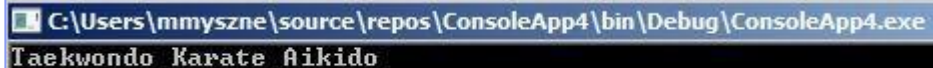
```
using System;
using System.Collections.Generic;

namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una lista de Strings usando inicializador de objetos
            var artesMarciales = newList<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.WriteLine(arte + " ");
}
}
```



```
C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Taekwondo Karate Aikido
```


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.

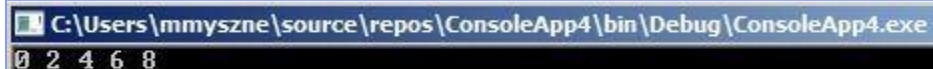
```
using System;
using System.Collections.Generic;

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



```
C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
0 2 4 6 8
```

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;

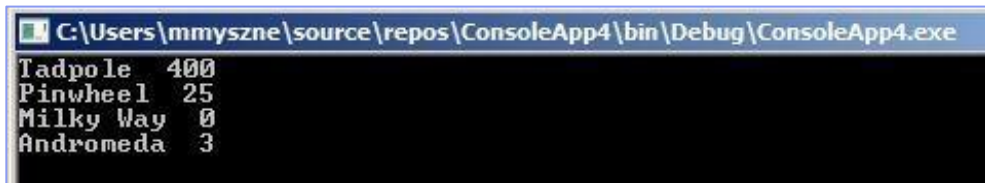
namespace ConsoleApp4
{
    class Program
    {
        public class Galaxia
        {
            public string Nombre { get; set; }
            public int MegaAniosLuz { get; set; }
        }
    }
}
```

...

```
static void Main(string[] args)
{
    vList<Galaxia> lasGalaxias = newList<Galaxia> {
        newGalaxia() { Nombre="Tadpole", MegaAniosLuz=400},
        newGalaxia() { Nombre="Pinwheel", MegaAniosLuz=25},
        newGalaxia() { Nombre="MilkyWay", MegaAniosLuz=0},
        newGalaxia() { Nombre="Andromeda", MegaAniosLuz=3}
    };

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





A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe. The window contains the following text output:

```
Tadpole 400  
Pinwheel 25  
Milky Way 0  
Andromeda 3
```

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.

```
enumDias{ 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.

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

En la enumeración del último ejemplo, 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 ve en el ejemplo que mostramos a continuación:

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

Tipos admitidos para una enumeración

Los tipos admitidos para una enumeración son los siguientes: **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** usando 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 conllevar a crear posibles problemas de versiones.

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 que vemos a continuación, 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);
    }
}
```

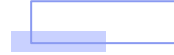
```
C:\Users\mmyszne\source\repos\ConsoleApp4\bin\Debug\ConsoleApp4.exe
Dom = 1
Vie = 6
```

En el ejemplo, la opción de tipo base se usa 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.



**¡Sigamos
trabajando!**