Módulo II

Programación II

Módulo III: Gestión Dinamica de Memoria

Módulo II:

- **Una pila en C#**
- **En cola C#**
- **Las listas.**
- **ArrayList.**
- **Los enumeradores.**
- **Introducción a los “Generics”**

Una pila en C#

Para crear una pila, emplearemos la clase Stack. Una pila nos permitirá introducir un nuevo elemento en la cima ("apilar", en inglés "push") y quitar el elemento que hay en la cima ("desapilar", en inglés "pop").

Este tipo de estructuras se suele denotar también usando las siglas "LIFO" (Last In First Out: lo último en entrar es lo primero en salir).

Para utilizar la clase "Stack" y la mayoría de las que veremos en este tema, necesitamos incluir en nuestro programa una referencia a "System.Collections".

Así, un ejemplo básico que creara una pila, introdujera tres palabras y luego las volviera a mostrar sería:

```
using System;
using System.Collections;

public class Ejemplo_11_02a
{
    public static void Main()
    {
        string palabra;

        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = (string) miPila.Pop();
            Console.WriteLine( palabra );
        }
    }
}
```

cuyo resultado sería:

```
yo
soy
Hola,
```

Como se puede ver en este ejemplo, no hemos indicado que sea una "pila de strings", sino simplemente "una pila". Por eso, los datos que extraemos son "objetos", que deberemos convertir al tipo de datos que nos interese utilizando un "typecast" (conversión forzada de tipos), como en

```
palabra  =  (string) miPila.Pop();
```

La implementación de una pila en C# es algo más avanzada que lo que podríamos esperar en una pila estándar: incorpora también métodos como:

- "Peek", que mira el valor que hay en la cima, pero sin extraerlo.
- "Clear", que borra todo el contenido de la pila.
- "Contains", que indica si un cierto elemento está en la pila.
- "GetType", para saber de qué tipo son los elementos almacenados en la pila.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la pila, una funcionalidad que veremos con algún detalle más adelante.
- También tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Una cola en C#

Podemos crear colas si nos apoyamos en la clase Queue. En una cola podremos introducir elementos por la cabeza ("Enqueue", encolar) y extraerlos por el extremo opuesto, el final de la cola ("Dequeue", desencolar). Este tipo de estructuras se nombran a veces también por las siglas FIFO (First In First Out, lo primero en entrar es lo primero en salir). Un ejemplo básico similar al anterior, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```
using System;
using System.Collections;

public class Ejemplo_11_03a
{
    public static void Main()
    {
        string palabra;

        Queue miCola = new Queue();
        miCola.Enqueue("Hola,");
        miCola.Enqueue("soy");
        miCola.Enqueue("yo");

        for (byte i=0; i<3; i++)
        {
            palabra = (string) miCola.Dequeue();
            Console.WriteLine( palabra );
        }
    }
}
```

que mostraría:

```
Hola,
soy
yo
```

Al igual que ocurría con la pila, la implementación de una cola que incluye C# es más avanzada que eso, con métodos similares a los de antes:

- "Peek", que mira el valor que hay en la cabeza de la cola, pero sin extraerlo.
- "Clear", que borra todo el contenido de la cola.
- "Contains", que indica si un cierto elemento está en la cola.
- "GetType", para saber de qué tipo son los elementos almacenados en la cola.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la cola, una funcionalidad que veremos con algún detalle más adelante.
- Al igual que en la pila, también tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Las listas

Una lista es una estructura dinámica en la que se puede añadir elementos sin tantas restricciones. Es habitual que se puedan introducir nuevos datos en ambos extremos, así como entre dos elementos existentes, o bien incluso de forma ordenada, de modo que cada nuevo dato se introduzca automáticamente en la posición adecuada para que todos ellos queden en orden.

En el caso de C#, no tenemos ninguna clase "List" que represente una lista genérica, pero sí dos variantes especialmente útiles: una lista a cuyos elementos se puede acceder como a los de un array ("ArrayList") y una lista ordenada ("SortedList").

ArrayList

En un ArrayList, podemos añadir datos en la última posición con "Add", insertar en cualquier otra con "Insert", recuperar cualquier elemento usando corchetes, o incluso ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
using System;
using System.Collections;

public class Ejemplo_11_04_01a
{
    public static void Main()
    {
        ArrayList miLista = new ArrayList();
        // Añadimos en orden
        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido actual:");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Accedemos a una posición
        Console.WriteLine( "La segunda palabra es: {0}",
            miLista[1] );

        // Insertamos en la segunda posicion
        miLista.Insert(1, "Como estas?");

        // Mostramos de otra forma lo que contiene
        Console.WriteLine( "Contenido tras insertar:");

        for (int i=0; i<miLista.Count; i++)
            Console.WriteLine( miLista[i] );

        // Buscamos un elemento
        Console.WriteLine( "La palabra \"yo\" está en la posición {0}",
            miLista.IndexOf("yo") );

        // Ordenamos
        miLista.Sort();

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido tras ordenar");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Buscamos con búsqueda binaria
        Console.WriteLine( "Ahora \"yo\" está en la posición {0}",
            miLista.BinarySearch("yo") );

        // Invertimos la lista
        miLista.Reverse();

        // Borramos el segundo dato y la palabra "yo"
        miLista.RemoveAt(1);
        miLista.Remove("yo");
```

```
        // Mostramos nuevamente lo que contiene
        Console.WriteLine( "Contenido dar la vuelta y tras eliminar dos:");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Ordenamos y vemos dónde iría un nuevo dato
        miLista.Sort();
        Console.WriteLine( "La frase \"Hasta Luego\"...");
        int posicion = miLista.BinarySearch("Hasta Luego");
        if (posicion >= 0)
            Console.WriteLine( "Está en la posición {0}", posicion );
        else
            Console.WriteLine( "No está. El dato inmediatamente mayor "+
                "es el {0}: {1}",
                ~posicion, miLista[~posicion] );
    }
}
```

El resultado de este programa es:

```
Contenido actual:
Hola,
soy
yo
La segunda palabra es: soy
Contenido tras insertar:
Hola,
Como estas?
soy
yo
La palabra "yo" está en la posición 3
Contenido tras ordenar
Como estas?
Hola,
soy
yo
Ahora "yo" está en la posición 3
Contenido dar la vuelta y tras eliminar dos:
Hola,
Como estas?
La frase "Hasta Luego"...
No está. El dato inmediatamente mayor es el 1: Hola,
```

Casi todo debería resultar fácil de entender, salvo quizá el símbolo ~. Esto se debe a que BinarySearch devuelve un número negativo cuando el texto que buscamos no aparece, pero ese número negativo tiene un significado: es el "valor complementario" de la posición del dato inmediatamente mayor (es decir, el dato cambiando los bits 0 por 1 y viceversa). En el ejemplo anterior, "posición" vale -2, lo que quiere decir que el dato no existe, y que el dato inmediatamente mayor está en la posición 1 (que es el "complemento a 2" del número -2, que es lo que indica la expresión "~posición"). En el apéndice 3 de este texto hablaremos de

cómo se representan internamente los números enteros, tanto positivos como negativos, y entonces se verá con detalle en qué consiste el "complemento a 2".

A efectos prácticos, lo que nos interesa es que si quisiéramos insertar la frase "Hasta Luego", su posición correcta para que todo el ArrayList permaneciera ordenado sería la 1, que viene indicada por "~posicion".

Veremos los operadores a nivel de bits, como ~, en el tema 13, que estará dedicado a otras características avanzadas de C#.

Los "enumeradores"

Un enumerador es una estructura auxiliar que permite recorrer las estructuras dinámicas de forma secuencial. Casi todas ellas contienen un método GetEnumerator, que permite obtener un enumerador para recorrer todos sus elementos. Por ejemplo, en una tabla hash podríamos hacer:

```
using System;
using System.Collections;

public class Ejemplo_11_06a
{
    public static void Main()
    {
        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IDictionaryEnumerator miEnumerador = miDiccio.GetEnumerator();
        while ( miEnumerador.MoveNext() )
            Console.WriteLine("{0} = {1}",
                miEnumerador.Key, miEnumerador.Value);
    }
}
```

cuyo resultado es

```
Contenido:
pc = personal computer
byte = 8 bits
kilobyte = 1024 bytes
```

Como se puede ver, los enumeradores tendrán un método "MoveNext", que intenta moverse al siguiente elemento y devuelve "false" si no lo consigue. En el caso de las tablas hash, que tiene dos campos (clave y valor), el enumerador a usar será un "enumerador de diccionario" (IDictionaryEnumerator), que contiene los campos Key y Value.

Como se ve en el ejemplo, es habitual que no obtengamos la lista de elementos en el mismo orden en el que los introdujimos, debido a que se colocan siguiendo la función de dispersión.

Para las colecciones "normales", como las pilas y las colas, el tipo de Enumerador a usar será un IEnumerator, con un campo Current para saber el valor actual:

```
using System;
using System.Collections;

public class Ejemplo_11_06b
{
    public static void Main()
    {
        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IEnumerator miEnumerador = miPila.GetEnumerator();

        while ( miEnumerador.MoveNext() )
            Console.WriteLine("{0}", miEnumerador.Current);
    }
}
```

que escribiría

```
Contenido:
yo
soy
Hola,
```

Nota: los "enumeradores" existen también en otras plataformas, como Java, aunque allí reciben el nombre de "iteradores".

Se puede saber más sobre las estructuras dinámicas que hay disponibles en la plataforma .Net consultando la referencia en línea de MSDN (mucha de la cual está sin traducir al español):

http://msdn.microsoft.com/es-es/library/system.collections(en-us,VS.71).aspx#

Introducción a los "generics"

Una ventaja, pero también a la vez un inconveniente, de las estructuras dinámicas que hemos visto, es que permiten guardar datos de cualquier tipo, incluso datos de distinto tipo en una misma estructura: un ArrayList que contenga primero un string, luego un número entero, luego uno de coma flotante, después un

struct… Esto obliga a que hagamos una "conversión de tipos" con cada dato que obtenemos (excepto con los "strings").

En ocasiones puede ser interesante algo un poco más rígido, que con las ventajas de un ArrayList (crecimiento dinámico, múltiples métodos disponibles) esté adaptado a un tipo de datos, y no necesite una conversión de tipos cada vez que extraigamos un dato.

Por ello, en la versión 2 de la "plataforma .Net" se introdujeron los "generics", que definen estructuras de datos genéricas, que nosotros podemos particularizar en cada uso. Por ejemplo, una **lista** de strings se definiría con:

```
List<string> miLista = new List<string>();
```

Y necesitaríamos incluir un nuevo "using" al principio del programa:

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

Ejemplo:

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

public class Ejemplo_11_08a
{
    public static void Main()
    {
        List<string> miLista = new List<string>();
        // Añadimos en orden
        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido actual:");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Accedemos a una posición
        Console.WriteLine( "La segunda palabra es: {0}",
            miLista[1] );

        // Insertamos en la segunda posicion
        miLista.Insert(1, "Como estas?");

        // Mostramos de otra forma lo que contiene
        Console.WriteLine( "Contenido tras insertar:");
        for (int i=0; i<miLista.Count; i++)
            Console.WriteLine( miLista[i] );

        // Buscamos un elemento
        Console.WriteLine( "La palabra \"yo\" está en la posición {0}",
            miLista.IndexOf("yo") );
```

```
// Ordenamos
miLista.Sort();

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido tras ordenar");
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Buscamos con búsqueda binaria
        Console.WriteLine( "Ahora \"yo\" está en la posición {0}",
            miLista.BinarySearch("yo") );

        // Invertimos la lista
        miLista.Reverse();

        // Borramos el segundo dato y la palabra "yo"
        miLista.RemoveAt(1);
        miLista.Remove("yo");

        // Mostramos nuevamente lo que contiene
        Console.WriteLine( "Contenido dar la vuelta y tras eliminar dos:");
        foreach (string frase in miLista)

            Console.WriteLine( frase );

        // Ordenamos y vemos dónde iría un nuevo dato
        miLista.Sort();
        Console.WriteLine( "La frase \"Hasta Luego\"...");
        int posicion = miLista.BinarySearch("Hasta Luego");
        if (posicion >= 0)
            Console.WriteLine( "Está en la posición {0}", posicion );
        else
            Console.WriteLine( "No está. El dato inmediatamente mayor "+
                "es el {0}: {1}",
                ~posicion, miLista[~posicion] );
    }
}
```

De esta misma forma, podríamos crear una lista de structs, o de objetos, o de cualquier otro dato.

No sólo tenemos listas. Por ejemplo, también existe un tipo "**Dictionary**", que equivale a una tabla Hash, pero en la que las claves y los valores no tienen por qué ser strings, sino el tipo de datos que nosotros decidamos. Por ejemplo, podemos usar una cadena como clave, pero un número entero como valor obtenido:

```
Dictionary<string, int> dict = new Dictionary<string, int>();
```

Así, con un diccionario que tenga tanto claves string como valores string, podríamos crear una versión alternativa del ejemplo 11_05b. Los únicos cambios serían una declaración parecida a la anterior, el "using" correcto, y cambiar Contains por ContainsKey:

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

public class Ejemplo_11_08b
{
    public static void Main()
    {
        // Creamos e insertamos datos
        Dictionary<string, string> miDiccio = new Dictionary<string,
string>();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );

        if (miDiccio.ContainsKey("pc"))
            Console.WriteLine( "El significado de PC es: {0}",
                miDiccio["pc"]);
        else
            Console.WriteLine( "No existe la palabra PC");
    }
}
```