C#

Gestión dinámica de memoria

Índice

Estructuras dinámicas de datos

Permiten almacenar un conjunto cambiante de datos complejos,no tienen un tamaño prefijado, se pueden añadir o quitar elementos durante la ejecución del programa.

Los tipos principales son:

-Listas, pilas y colas

-Diccionarios o mapas

-Conjuntos

Pilas

Se crean con la clase **Stack** del espacio **System.Collections**

Permiten añadir y quitar elementos sólo por un extremo llamado tope.

Estructuras **LIFO** (Last In, First Out).

Métodos útiles

Push(elemento): añade el elemento al tope.

Pop(): quita el elemento del tope y lo devuelve.

Peek(): consulta el elemento del tope (sin quitarlo).

Count: obtiene cuántos elementos hay en la pila.

Clear(): elimina todos los elementos de la pila.

Contains(elemento): comprueba si la pila contiene al elemento.

ToArray(): obtiene un array con los elementos de la pila.

```
using System;
using System.Collections;

class PruebaPila
{
   static void Main()
   {
      Stack pila = new Stack();
      pila.Push("Uno");
      pila.Push("Dos");
      pila.Push("Tres");

      Console.WriteLine(pila.Peek());  // Tres
      while(pila.Count > 0)
      {
         string dato = (string)pila.Pop();
         Console.WriteLine(dato);
      }
   }
}
```

Genéricos(Generics)

Permiten especificar de qué tipo es una colección concreta.

Facilitan la gestión de elementos de la colección (evitan **typecast**).

Espacio de nombres **System.Collections.Generic** en lugar de **System.Collections.**

Especificamos entre < y > el tipo de dato de la colección.

Pila usando genéricos

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

class PruebaPila
{
   static void Main()
   {
      Stack<string> pila = new Stack<string>();
      pila.Push("Uno");
      pila.Push("Dos");
      pila.Push("Tres");

      Console.WriteLine(pila.Peek());  // Tres
      while(pila.Count > 0)
      {
         string dato = pila.Pop();
         Console.WriteLine(dato);
      }
   }
}
```

Colas

Se crean con la clase **Queue** del espacio **System.Collections** (o **System.Collections.Generic** si queremos colas genéricas).

Permiten añadir elementos por un extremo y quitarlos por el opuesto.

Estructuras **FIFO** (First In, First Out).

Métodos útiles

Enqueue(elemento): añade el elemento al final.

Dequeue(): quita el elemento del inicio de la cola y lo devuelve.

Resto de métodos similares a la pila(**Peek()**, **Count**, **Clear()**, **Contains(…)**,etc.

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

class PruebaCola
{
   static void Main()
   {
      Queue<string> cola = new Queue<string>();
      cola.Enqueue("Uno");
      cola.Enqueue("Dos");
      cola.Enqueue("Tres");

      Console.WriteLine(cola.Peek());  // Uno
      while(cola.Count > 0)
      {
         string dato = cola.Dequeue();
         Console.WriteLine(dato);
      }
   }
}
```

Listas

Colecciones indexadas de datos(Cada elemento ocupa una posición numérica empezando por 0, como los arrays).

Podemos añadir/quitar libremente elementos en cualquier posición.

Métodos de interés

Add(elemento): añade el elemento al final de los existentes.

Insert(posicion, elemento): inserta el elemento en la posición indicada (y desplaza a la derecha el resto de elementos).

IndexOf(elemento): devuelve la posición donde se encuentra el elemento, o -1 si no existe.

RemoveAt(posicion): elimina el elemento de la posición indicad (y desplaza el resto a la izquierda).

Count, **Clear()**, **Contains(...)** igual que las anteriores.

Borrados en listas

Cuando tengamos que eliminar un elemento de la lista, debemos tener en cuenta que:

-Al borrar una posición, el resto de posiciones a la derecha se desplaza a la izquierda automáticamente.

-No conviene usar bucles tipo for para borrar elementos de forma masiva (corremos el riesgo de dejar alguno sin borrar al desplazarse la lista).

Ordenación de listas de objetos

Podemos ordenar listas de objetos haciendo que la clase de esos objetos defina el criterio de comparación implementando la interfaz **Icomparable**.

Alternativamente, también podemos utilizar una clase adicional que implemente la interfaz **IComparer**.

Llamamos al método **Sort** de la lista y queda automáticamente ordenada en base al criterio definido.

Icomparable

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

class Persona : IComparable<Persona>
{
    private string nombre;
    private int edad;

    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public override string ToString()
    {
        return nombre + " (" + edad + " años)";
    }

    public int CompareTo(Persona otra)
    {
        return otra.edad.CompareTo(this.edad);
    }
}
```

```
class Principal
{
    static void Main()
    {
        List<Persona> personas = new List<Persona>();
        personas.Add(new Persona("Nacho", 43));
        personas.Add(new Persona("Ana", 39));
        personas.Add(new Persona("Juan", 70));
        personas.Add(new Persona("Mario", 8));

        personas.Sort();

        foreach(Persona p in personas)
        {
            Console.WriteLine(p);
        }
    }
}
```

IComparer

```
// Nueva clase que implementa la interfaz
class ComparadorPersonas: IComparer<Persona>
{
    public int Compare(Persona p1, Persona p2)
    {
        return p1.Nombre.CompareTo(p2.Nombre);
    }
}

// Clase Persona (sin interfaz)
class Persona
{
    private string nombre;
    private int edad;

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }

    public int Edad
    {
        get { return edad; }
        set { edad = value; }
    }
```

```
    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public override string ToString()
    {
        return nombre + " (" + edad + " años)";
    }
}

class Principal
{
    static void Main()
    {
        List<Persona> personas = new List<Persona>();
        personas.Add(new Persona("Nacho", 43));
        personas.Add(new Persona("Ana", 39));
        personas.Add(new Persona("Juan", 70));
        personas.Add(new Persona("Mario", 8));

        personas.Sort(new ComparadorPersonas());

        foreach(Persona p in personas)
            Console.WriteLine(p);
    }
}
```

Hay 2 tipos:

ArrayList

Lista que internamente contiene un array con las posiciones.

No tenemos que preocuparnos de la gestión del array.

Usamos los corchetes para acceder a la posición deseada (como los arrays).

List

Es igual que **ArrayList** pero permite el uso de genéricos.

ArrayList

```
using System;
using System.Collections;

class PruebaArrayList
{
   static void Main()
   {
      ArrayList lista = new ArrayList();
      lista.Add("Uno");
      lista.Add("Tres");
      lista.Add("Cuatro");
      lista.Insert(1, "Dos");

      Console.WriteLine(lista[2]);    // Tres

      for (int i = 0; i < lista.Count; i++)
      {
         string dato = (string)lista[i];
         Console.WriteLine(dato);
      }
   }
}
```

List

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

class PruebaList
{
   static void Main()
   {
      List<string> lista = new List<string>();
      lista.Add("Uno");
      lista.Add("Tres");
      lista.Add("Cuatro");
      lista.Insert(1, "Dos");

      Console.WriteLine(lista[2]);    // Tres

      for (int i = 0; i < lista.Count; i++)
      {
         string dato = lista[i];
         Console.WriteLine(dato);
      }
   }
}
```

Diccionarios

Estructuras dinámicas donde los datos no se asocian a una posición numérica, sino a una clave.

Sabiendo la clave, se accede directamente al dato asociado, sin necesidad de recorrer toda la colección.

Hay 2 tipos:

Listas ordenadas (Sorted list)

Accedemos a los elementos poniendo su clave entre corchetes.

Los datos se ordenan automáticamente por su clave.

Add(clave, valor) añade un nuevo valor a la lista, asociado a la clave indicada.

Si la clave existe, se produce una excepción.

Remove(clave) elimina un valor de la lista a partir de su clave.

Contains(clave) determina si existe la clave en el diccionario. **ContainsKey** en genéricos.

Count determina el tamaño del diccionario.

```
using System;
using System.Collections;

class EjemploSortedList
{
   static void Main()
   {
      SortedList terminos = new SortedList();
      terminos["one"] = "uno";
      terminos["two"] = "dos";
      terminos["hello"] = "hola";

      Console.WriteLine(terminos["two"]); // dos
   }
}
```

Usando genéricos

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

class EjemploSortedList
{
   static void Main()
   {
      SortedList<string, string> terminos = new
SortedList<string, string>();
      terminos["one"] = "uno";
      terminos["two"] = "dos";
      terminos["hello"] = "hola";

      Console.WriteLine(terminos["two"]); // dos
   }
}
```

Recorrido

Hacemos un bucle for normal, y sacamos la clave y el valor de cada posición.

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

class EjemploSortedList
{
   static void Main()
   {
      SortedList<string, string> terminos = new
SortedList<string, string>();
      terminos["one"] = "uno";
      terminos["two"] = "dos";
      terminos["hello"] = "hola";

      for (int i = 0; i < terminos.Count; i++)
      {
         Console.WriteLine(terminos.Keys[i] + " = " +
terminos.Values[i]);
         // Estos métodos se llaman GetKey(i) y
GetByIndex(i) si no es genérica
      }
   }
}
```

Tablas hash (Hashtable o Dictionary)

Similares a las **sorted lists**, pero se accede más rápidamente a los elementos del diccionario.

Esto se consigue automáticamente con una función de **hashing** que asocia a cada clave un número diferente, con el que localizar el dato en la colección(No tenemos que preocuparnos de dicha función, en general).

Usamos la clase **Hashtable** (no genérica) o **Dictionary** (genérica), con métodos similares a **SortedList**.

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

class Libro
{
   private string isbn;
   private string titulo;
   private int paginas;

   ...
}

class EjemploHash
{
   static void Main()
   {
      Dictionary<string, Libro> catalogo = new
Dictionary<string, Libro>();
      catalogo.Add("1234112", new Libro("1234112", "El
juego de Ender", 323));
      catalogo.Add("4425353", new Libro("4425353", "La
tabla de Flandes", 356));

      Console.WriteLine(catalogo["1234112"].GetTitulo());

      foreach(KeyValuePair<string, Libro> dato in catalogo)
      {
         Console.WriteLine(dato.Key + " = " +
dato.Value.GetTitulo());
      }
   }
}
```

Enumeradores

Permiten recorrer distintos tipos de colecciones.

IEnumerator para listas, pilas, colas.

IDictionaryEnumerator para diccionarios.

Necesitamos **System.Collections.**

Creamos el enumerador sobre la colección.

Usamos métodos como **MoveNext()** o propiedades como **Current**, **Key** o **Value** para recorrer los datos y sacar la información.

Ejemplo de recorrido de pila

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

class PruebaPila
{
   static void Main()
   {
      Stack<string> pila = new Stack<string>();
      pila.Push("Uno");
      pila.Push("Dos");
      pila.Push("Tres");

      IEnumerator enumerador = pila.GetEnumerator();
      while (enumerador.MoveNext())
      {
         Console.WriteLine(enumerador.Current);
      }
   }
}
```

Ejemplo de recorrido de diccionario

```
class Libro
{
   string isbn;
   string titulo;
   int paginas;

   ...
}

class EjemploHash
{
   static void Main()
   {
      Dictionary<string, Libro> catalogo = new
Dictionary<string, Libro>();
      catalogo.Add("1234112", new Libro("El juego de
Ender", 323));
      catalogo.Add("4425353", new Libro("La tabla de
Flandes", 356));

      IDictionaryEnumerator enumerador =
catalogo.GetEnumerator();
      while(enumerador.MoveNext())
      {
         Console.WriteLine(enumerador.Key + " = " +
enumerador.Value);
      }
   }
}
```

Conjuntos

Colecciones dinámicas de datos que no admiten elementos repetidos.

Alternativas:

SortedSet: similar a **SortedList** en cuanto a que permite obtener los datos en orden.

HashSet: similar a **Hashtable** a la hora de acceder rápidamente a los valores.

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

class PruebaConjuntos
{
   static void Main()
   {
      HashSet<string> datos = new HashSet<string>();
      datos.Add("Uno");
      datos.Add("Uno");   // No tiene efecto, no se añade
      datos.Add("Dos");

      foreach(string s in datos)
      {
         Console.WriteLine(s);
      }
   }
}
```

Conjuntos y datos complejos

Si creamos un conjunto de datos complejos (personas, libros, etc) C# no sabe cómo detectar duplicados.

Debemos indicarle nosotros el criterio de igualdad implementando los métodos **Equals** y **GetHashCode**.

```
class Persona
{
    private string nombre;
    private int edad;

    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public override string ToString()
    {
        return nombre + ", " + edad + " años";
    }
}

...

HashSet<Persona> personas = new
HashSet<Persona>();
personas.Add(new Persona("Nacho", 43));
personas.Add(new Persona("Ana", 39));
personas.Add(new Persona("Nacho", 43));
personas.Add(new Persona("Pepe", 70));
personas.Add(new Persona("Ana", 39));

Console.WriteLine(personas.Count);      // Saca 5
```

Equals y GetHashCode

Equals determina en base a qué atributos son iguales dos objetos.

GetHashCode genera un código hash que será el mismo para los atributos que indiquemos(Dos objetos que coincidan en esos atributos tendrán el mismo código y serán iguales a efectos prácticos).

Son necesarios los dos: C# comprueba si el código hash es el mismo, y si los atributos son los mismos.

```
class Persona
{
    ...
    public override bool Equals(object? obj)
    {
        return obj is Persona persona &&
               nombre == persona.nombre &&
               edad == persona.edad;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(nombre, edad);
    }
}
```