Tema 10.1

Descargar estos apuntes

Índice

Colecciones en la BCL

Una **colección** es un tipo de dato cuyos objetos almacenan otros objetos. Un ejemplo típico son las tablas, aunque en la BCL se incluyen muchas otras clases de colecciones que iremos viendo a lo largo de este tema. En las versiones recientes de c# las podemos encontrar en System.Collections.Generic.

Aunque las colecciones predefinidas incluidas en la BCL disponen de miembros propios con los que manipularlas, todas incluyen al menos los miembros de **ICollection**. En realidad la interfaz **ICollection** hereda de la interfaz **IEnumerable** en que se basa la instrucción foreach. Y también implementan la interfaz **ICIonable**, formada por un único método object Clone() que devuelve una copia del objeto sobre el que se aplica.

Algunas de las colecciones más utilizadas son las siguientes:

- Las listas que implementan el interfaz IList. Entre ellas se encuentran los **Arrays** tradicionales, sólo que tendremos restricciones de añadir, remover, etc…
  ArrayList aparece en la v1.0 e implementa IList completo permitiendo Añadir y Borrar e incluyendo métodos como BinarySearch y Sort que a diferencia de sus hermanos en los arrays no son estáticos.
  List<T> es el equivalente genérico a ArrayList y será el que utilizaremos en su lugar, por ser más robusta en la restricción de tipos. Otro tipo sería SortedList que es una lista ordenada, pero no la veremos en este tema.
- También ofrece de la colección **LinkedList<T>**, que no implementa la interfaz IList y se basa en el concepto de nodo, visto anteriormente.
- Las **HashTable** o Dispersión y los **Dictionary<T>**. son estructuras de datos que asocian una clave a un valore. La operación principal que soporta de manera más eficiente, es la búsqueda.
- Otros tipos menos usados son
  las Pilas **Stack** o **Stack<T>** y las Colas **Queue** o **Queue<T>**. También se implementan como tablas, sólo que en las colas podemos establecer un factor de crecimiento al añadir.

Colección List<T>

Las listas son una especie de arrays dinámicos que permiten añadir o quitar elementos en cualquier momento y de cualquier parte del array. Para trabajar con ellas se utiliza el TAD List<T> de C#.

```
List<T> lista=new List<T>()
```

Al crearnos una lista de tipo Persona podremos acceder a sus miembros. Lo podemos ver en el siguiente ejemplo:

```
struct Persona
{
  public string Nombre;
  public int Edad;
}
class Program
{
  static void Main(string[] args)
  {
    List<Persona> personas = new List<Persona>();
    Persona p;
    for (int i = 0; i < 5; i++)
    {
      p.Nombre = Console.ReadLine();
      p.Edad = int.Parse(Console.ReadLine());
      personas.Add(p);
    }
    for (int i = 0; i <personas.Count ; i++)
      Console.WriteLine(personas[i].Nombre);
  }
}
```

Principales operaciones que se pueden realizar en una lista:

- **Inicializar** la Lista, para poder usarla, lo primero que debemos hacer es crear la lista (vacía), para luego ir añadiendo elementos. Esto se hace definiendo una variable de tipo List y usando el operador new para inicializarla, de la siguiente manera:
  `List<string> lista =new List<string>()`
- **Añadir** datos una vez tenemos la lista creada, podemos añadir datos de dos formas:
  - Usando el método **Add** de la propia lista, añadiremos un elemento al final de ella.
    `lista.Add("hola")`
  - Usando el método Insert de la propia lista, añadiremos un elemento en la posición que indiquemos.
    `lista.Insert(2, "otra cadena")`

A la hora de gestionar las listas, hay que tener en cuenta que las posiciones empiezan a numerarse por el cero.

- **Eliminar** un dato de la lista, en este caso también tenemos varias alternativas:
  - Usando el método **Remove** de la propia lista, eliminamos el elemento que coincida con el pasado como parámetro.
    `lista.Remove("hola")`
    Elimina el elemento "hola" de la lista. Si hubiera varios, elimina el primero que se encuentre.
  - Usando el método **RemoveAt** de la propia lista, elimina el elemento de la posición que indiquemos:
    `lista.RemoveAt(2)`
- **Modificar** un dato. Si queremos cambiar directamente el valor de uno de los datos de la lista, basta con que accedamos a su posición y modifiquemos o le asignemos otro valor. Por ejemplo:
  `Lista[1]="Hola"`
- **Recorrer** los elementos de la lista a través de un bucle for o forech. La longitud de la lista, nos la da la propiedad **Count**.

```
for (int i=0; i< lista.Count; i++) Console.WriteLine(lista[i])
foreach (var c in lista) Console.WriteLine(c)
```

Colección Dictionary<T>

Las tablas hash o Diccionarios son otro tipo de colecciones con un comportamiento particular. Hasta ahora, todos los elementos de una colección tienen una posición numérica. Si tenemos una lista, el primer elemento ocupa la posición 0, el siguiente la 1, etc. Si queremos acceder al cuarto elemento de una lista llamada miLista, tenemos que poner miLista[3], y si no sabemos la posición debemos usar un bucle.

Para evitar el bucle y realizar un acceso directo podemos usar tablas hash. En este tipo de colecciones, cada dato que agregamos a ella no tiene asociado una posición numérica, sino una clave que lo identifique. De esta manera, si conocemos la clave del elemento, podemos acceder directamente a sus datos sin tener que recorrer toda la lista.

Por ejemplo, podemos asociar el dni de cada persona con sus datos completos, teniendo al final una tabla como esta:

Clave	Valor
11224441K	Nombre="Pepe" Edad = 30
11335499M	Nombre="María" Edad = 22
12345678O	Nombre="Juan" Edad = 33

```
Si quiero consultar los datos de María, buscaré por su clave que es su dni. También notar que la clave puede ser cualquier tipo de dato. En
```

El funcionamiento es similar a un diccionario. Si quiero consultar el significado de una palabra y sé cuál es esa palabra, voy a la página donde está y la consulto, sin tener que ir palabra a palabra comprobando si es esa la que busco.

Las tablas hash en C# se manejan con el TAD Hashtable, no genérico y con el TAD Dictionary<T>, este último más usado.

- **Inicializar** un diccionario mediante el constructor del tab Dictionary.
  `Dictionary<TClave, TValor> tabla=new Dictionary<TClave, TValor>()`
- **Añadir** un dato al diccionario creado, se puede realizar de varias formas, pero lo más normal es usar el método Add, indicando la clave que queremos asociar a cada elemento y el elemento en sí.
  Si por ejemplo estamos haciendo una tabla de elementos de tipo Persona, la clave puede ser el dni de la persona en sí, y el elemento a guardar el resto de datos:

```
static void Main(string[] args)
{
   Dictionary<String, Persona> p = new Dictionary<string, Persona>();
   Persona persona=new Persona();
   for (int i = 0; i < 5; i++)
   {
       persona.Nombre = ("nombre" + i);
       persona.Edad = 10 + i;
       Console.WriteLine("Introduce un DNI para{0}", persona.Nombre);
       p.Add(Console.ReadLine(), persona);
   }
   Console.WriteLine("Introduce un DNI:");
   string dni = Console.ReadLine();
   Console.WriteLine(p[dni].Edad);
   p.Remove(dni);
   foreach (string x in p.Keys) Console.Write(p[x].Edad);
}
```

- **Eliminar** un dato de la lista, para ello usamos el método **Remove** con la clave del elemento que queremos eliminar como argumento.
  `tabla.Remove("11223314L")`
- **Modificar** el valor de un dato almacenado en el diccionario. También se accede mediante la clave y se asigna el objeto con los cambios.
  `tabla["11224441K"]=new Persona("Pepe",31)`
- **Recorrer** diccionarios no es lo común, ya que su acceso es mediante clave. Aunque se puede realizar el acceso a todos los elementos usando foreach.
  Por ejemplo, este bucle saca las edades de todas las personas:
  `foreach (string x in p.Keys) Console.WriteLine(p[x].Edad)`
  Aunque, es posible que el orden no sea el mismo que cuando se introdujeron los datos, ya que las tablas hash tienen un mecanismo de ordenación diferente.

> **Nota:** si se intenta acceder a un elemento del Dictionary del que no existe la clave,el sistema lanzará una excepción. Por lo que es buena práctica utilizar el método `ContainsKey(clave)` , para comprobar si existe la clave antes de acceder al elemento a través de esta.

```
static void Main(string[] args)
{

    // Definimos el diccionario con los países y sus capitales.
    Dictionary<string, string> capitalesPorPais = new Dictionary<string, string>()
    {{"España", "Madrid"},{"Portugal", "Lisboa"}, {"Francia", "Paris"},
    {"Luxemburgo", "Luxemburgo"}, {"Irlanda", "Dublin"}};
    // Aunque hemos definido por extensión. Podemos añadir elemetos a posteriori.
    capitalesPorPais.Add("Belgica", "Bruselas");
    capitalesPorPais.Add("Alemania", "Berlin");
    // Obtenemos una lista de claves indizable por un entero.
    List<string> paises = new List<string>(capitalesPorPais.Keys);
    // Lista donde almacenaré los países ya preguntados para no repetirnos
    List<string> paisesPreguntados = new List<string>();

    const int NUMERO_PREGUNTAS = 5;
    Random semilla = new Random();
    uint puntos = 0;
    for (int i = 0; i < NUMERO_PREGUNTAS; i++)
    {
        string paisPreguntado;
        do
        {
           paisPreguntado = paises[semilla.Next(0, paises.Count)];
        } while (paisesPreguntados.Contains(paisPreguntado) == true);
        paisesPreguntados.Add(paisPreguntado);
        Console.Write($"¿Cual es la capital de {paisPreguntado}? > ");
        string capitalRespondida = Console.ReadLine().ToUpper();
        string mensaje;
        if (capitalRespondida == capitalesPorPais[paisPreguntado].ToUpper())
        {
            puntos += 2;
            mensaje = $"Correcto !!";
        }
        else
        {
            mensaje = "Incorrecto !!\n" +
                      $"La respuesta es {capitalesPorPais[paisPreguntado]}.";
        }
        mensaje += $"\nLlevas {puntos} puntos.\n";
                   Console.WriteLine(mensaje);
    }
    Console.WriteLine($"Tu nota final es {puntos}.");
}
```