

Arrays y colecciones

Arrays

Introducción.

En C# un array es un dato de tipo referencia. Un array es una estructura de datos que contiene variables (elementos) a los que se accede a través de índices.

Todos los elementos de un array son del mismo tipo. Se dice entonces que ése es el tipo del array. Así, se habla por ejemplo, de un *array de enteros* o un *array de string*, etc, dependiendo de que los elementos del array sean de un tipo o de otro.

Un array puede tener más de una dimensión. Por cada dimensión de un array se necesita un nivel de índices para acceder a sus elementos.

Para *declarar* un array se especifica el tipo de dato de los elementos seguido de unos corchetes vacíos y del nombre del array que es una referencia al objeto array. Por ejemplo:

```
int[] unArray;
```

La línea anterior declara `unArray` como una referencia a un array de enteros. Después de esta línea `unArray` es una referencia que contiene `null`, ya que no referencia a ningún array.

Como cualquier otro objeto, para crear un array de cualquier tipo se utiliza el operador `new`, que devuelve un array del número de elementos que se especifique en el número entre corchetes. Así, para crear un array de `string` de 100 elementos, se escribe:

```
string[] unString;  
unString = new string[100];
```

o bien, en una sola línea:

```
string[] unString = new string[100];
```

Aquí, se ha creado –con `new`– un objeto que es un array de 100 `string`, se declara una referencia al array –`unString`–, y se asigna el objeto a la referencia.

En un array de `N` elementos el primero tiene índice 0 y el último `N-1`. Para acceder a un elemento cualquiera, se utiliza el índice. Por ejemplo, para almacenar el valor 12 en el primer elemento de un array, se puede escribir:

```
//define un array de enteros de 23 elementos  
int[] unArray=new int[23];  
//almacena 12 en el primer elemento del array  
unArray[0]=12;
```

La sintaxis de la declaración e inicialización de un array es bastante flexible. Se puede, como se ha hecho anteriormente, declarar un array sin inicializarlo, y posteriormente inicializarlo.

Por ejemplo:

```
string[] unArray;  
...  
//Código  
...  
unArray=new string[3];  
unArray[0]="Eduardo";  
unArray[1]="Borja";  
unArray[2]="Gabriel";
```

También es posible declararlo e inicializarlo directamente, sin utilizar el operador `new`:

```
string[] unArray={"Eduardo","Borja","Gabriel"};
```

Lo anterior es equivalente a :

```
string[] unArray= new string[]{"Eduardo","Borja","Gabriel"};
```

o bien

```
string[] unArray= new string[3>{"Eduardo","Borja","Gabriel"};
```

El número de elementos que define la longitud de un array debe ser constante. Por eso, no es posible inicializar un array de esta manera:

```
int unEntero=3;  
string [] unArray= new string[unEntero>{"Eduardo","Borja","Gabriel"};
```

Para poder compilar, es necesario cambiar la primera de las dos líneas. Se ha de sustituir por la línea:

```
const int unEntero=3;
```

Los arrays son objetos

En C#, los arrays son objetos. Además, todos los arrays, de cualquier tipo derivan de la clase `System.Array`, que es el tipo base abstracto de todos los tipos de arrays.

La clase `System.Array` proporciona muchos métodos y propiedades para manipular arrays. Se pueden utilizar las propiedades y otros miembros que tiene esta clase para manipular los arrays. Por ejemplo, para obtener la longitud de un array se usa la propiedad `Length`. El código siguiente asigna la longitud del array `unArray`, que es 5, a la variable `longitudDelArray`:

```
int[]unArray = {1, 2, 3, 4, 5};  
int longitudDelArray = unArray.Length;
```

Es importante tener en cuenta que el tipo `System.Array` no es un tipo array, sino un tipo del cual derivan todos los tipos array.

Las propiedades y métodos más importantes son las siguientes:

Nota: En los ejemplos se utilizará el siguiente array de enteros:

```
int [] unArray=new int[]{2,5,3,4};
```

- `Length`: devuelve el número de elementos de un array. Por ejemplo:

```
int longitud=unArray.Length;
```

- `GetLength`: obtiene la longitud de una determinada dimensión cuando el array es multidimensional. Por ejemplo, para obtener el número de elementos de la primera dimensión:

```
int longitud=otroArray.GetLength(0);
```

- `Sort` y `Reverse`: Estos métodos permiten ordenar un array de modo ascendente o descendente. Son métodos estáticos. Por ejemplo:

```
Array.Sort(unArray);
```

o bien:

```
Array.Reverse (unArray);
```

En el siguiente ejemplo, se define un array de nombres y se ordenan:

```
//Se crea el array
string[] nombres={"Cristina","Matilde","Mª del Valle","Mª Teresa"};

//Se ordenan alfabéticamente dichos nombres
Array.Sort(nombres);

//Se imprimen los nombres ordenados
for(int i=0;i<nombres.Length;i++)
    Console.WriteLine(nombres[i]);
```

La salida de este programa es:

```
Cristina
Matilde
Mª del Valle
Mª Teresa
```

Por otro lado, aunque es una clase abstracta, el valor de una referencia de tipo `System.Array` en tiempo de ejecución puede ser `null` o la referencia a una instancia de un tipo array. `System.Array` no tiene un alias.

Arrays de varias dimensiones.

C# soporta arrays simples –de una dimensión-, multidimensionales y arrays de arrays.

Declaración de arrays

La sintaxis para declarar un array es la siguiente:

Array de una dimensión:

```
int[] unArray;
```

Array multidimensional o matriz rectangular:

```
string[,] arrayMultidimensional;  
int[, ,] botones
```

Array de arrays:

```
byte[][] arrayDeBytes;
```

Cuando se declara un array, sólo se define una referencia que puede apuntar a un array de ese tipo, pero que apunta a `null`. En C# los arrays son objetos y para que existan físicamente, debe crearse un objeto -deben ser instanciados-. La sintaxis para crear un array es:

o Array de una dimensión:

```
int[] unArray = new int[5];
```

Con esta sentencia, se crea una referencia a un array de enteros, se crea un array de 5 enteros –el objeto- y posteriormente se asigna la referencia anteriormente creada al array.

o Array multidimensional:

```
string[,] unaMatriz = new string[5,4];  
int[, ,] otraMatriz = new int[4,5,3];
```

o Array de arrays:

```
byte[][] arrayDeBytes = new byte[5][];  
for (int x = 0; x < arrayDeBytes.Length; x++)  
{  
    arrayDeBytes [x] = new byte[4];  
}
```

Se pueden mezclar matrices rectangulares y arrays de arrays. Por ejemplo:

```
int[ ][, ,][,] numeros;
```

Inicialización de arrays

C# proporciona una manera simple de declarar, crear e inicializar un array simultáneamente, incluyendo los valores iniciales entre llaves {}. Si no se hace así, el compilador lo inicializa automáticamente al valor por defecto.

A continuación se muestran distintos caminos para inicializar diferentes tipos de arrays.

o Array de una dimensión

```
int[] unArray = new int[5] {1, 2, 3, 4, 5};
string[] otroArray = new string[3] {"Pablo", "Arantza", "Marco"};
```

- Puede omitirse el tamaño del array cuando se inicializa:

```
int[] unArray = new int[] {1, 2, 3, 4, 5};
string[] otroArray = new string[] {"Pablo", "Arantza", "Marco"};
```

- Se puede omitir new si se inicializa:

```
int[] unArray = {1, 2, 3, 4, 5};
string[] otroArray = {"Pablo", "Arantza", "Marco"};
```

o Array multidimensional:

```
int[,] num = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] par=new string[2, 2]{{"Juan","Ana"}, {"Alberto","Maria"} };
```

- Se puede omitir el tamaño:

```
int[,] num = new int[,] { {1, 2}, {3, 4}, {5, 6} };
string[,] par = new string[,] { {"Juan","Ana"}, {"Alberto","Maria"} };
```

- También la cláusula new si se inicializa:

```
int[,] num = { {1, 2}, {3, 4}, {5, 6} };
string[,] par = { {"Juan","Ana"}, {"Alberto","Maria"} };
```

o Arrays de arrays

- Se puede inicializar de la siguiente forma:

```
int[][] num = new int[2][] {new int[] {2,3,4},new int[] {5,6,7,8,9} };
```

- u omitir el tamaño:

```
int[][] nums = new int[][] {new int[] {2,3,4},new int[] {5,6,7,8,9}};
```

Acceso a los miembros de un array

Se accede a los elementos de un array de manera similar a como se hace en C y C++. Por ejemplo, para acceder al quinto elemento del array y asignarle 27:

```
int[] unArray = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
unArray[4] = 27;
```

El siguiente código declara un array multidimensional y asigna 27 al elemento localizado en [1, 1]:

```
int[,] otroArray = { {1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10} };
otroArray[1, 1] = 27;
```

Con los arrays de arrays el acceso es como sigue:

```
int[][] tercerArray = new int[][]
{
    new int[] {1, 2},
    new int[] {3, 4}
};
tercerArray[1][1] = 5;
```

C# permite dos tipos de array multidimensionales: los arrays **rectangulares** y los arrays **dentados**.

Un **array bidimensional rectangular** es aquel en el cual todas sus filas tienen el mismo número de columnas. Los arrays que anteriormente se han mostrado como ejemplos, son rectangulares. En general son sencillos de declarar, inicializar y manejar. A continuación se declara un array de `string` bidimensional de cuatro filas y dos columnas:

```
string [,] misNombres={ { "Borja", "Yosune" },
                          { "Gabriel", "Patricia" },
                          { "Marco", "Arantxa" },
                          { "Eduardo", "Lola" } };
```

La otra alternativa de arrays multidimensionales es el **array dentado**, es decir, un array que no es rectangular. Un array bidimensional dentado, es un array en el que cada fila tienen un número diferente de columnas. Aunque evidentemente proporciona una mayor flexibilidad que el array rectangular, sin embargo son más complicados de declarar e inicializar. Para crear un array dentado, básicamente lo que se hace es crear un array de arrays:

Por ejemplo:

```
int[][] a=new int[3][];
a[0]=new int[4];
a[1]=new int[3];
a[2]=new int[1];
```

En lugar de usar la coma en el corchete, lo que se hace es usar un corchete extra, por cada dimensión. Por ejemplo, para declarar un array dentado de tres dimensiones:

```
int [][][] unArrayDeEnteros;
```

Un ejemplo completo puede aclarar esta idea:

```
string [][]animales=new string [3][];
animales[0]=new string[]{"Ballena","Tiburón","Delfín","Pulpo"};
animales[1]=new string[]{"Gorrión","Loro","Buitre"};
animales[2]=new string[]{"Perro","Gato"};

//Se imprime
for(int i=0;i<animales.GetLength(0);i++)
    for(int j=0 ; j< animales[i].Length ; j++)
        Console.WriteLine(animales [i][j]);
```

Ejemplo:

```
// arrays.cs
using System;
class EjemploDeDeclaracionDeArrays
{
    public static void Main()
    {
        //array de una dimensión
        int[] unArray = new int[5];
        Console.WriteLine("La longitud de unArray es {0}",
            unArray.Length);

        //array Multidimensional
        string[,] otroArray = new string[5,4];
        Console.WriteLine("La longitud de otroArray es {0}",
            otroArray.Length);

        // Array de arrays
        byte[][] arrayDeArrays = new byte[5][];
        Console.WriteLine("La longitud de arrayDeArrays es {0}",
            arrayDeArrays.Length);

        // crea un array de arrays
        for (int i = 0; i < arrayDeArrays.Length; i++)
        {
            arrayDeArrays[i] = new byte[i+3];
        }

        for (int i = 0; i < arrayDeArrays.Length; i++)
        {
            Console.WriteLine("La longitud de {0} es {1}", i,
                arrayDeArrays[i].Length);
        }
    }
}
```

La salida que se obtiene al ejecutar este programa es:

```
La longitud de unArray es 5
La longitud de otroArray es 20
La longitud de arrayDeArrays es 5
La longitud de 0 es 3
La longitud de 1 es 4
La longitud de 2 es 5
La longitud de 3 es 6
La longitud de 4 es 7
```

Usando foreach en Arrays

Se puede fácilmente recorrer un array con la sentencia `foreach`. Esta sentencia proporciona una manera elegante y sencilla de “recorrer” los distintos elementos de un array. Se utiliza para recorrer más fácilmente la colección o el array pero no debería utilizarse para cambiar los contenidos de la colección o array porque puede tener efectos colaterales no deseados. Por ejemplo, el siguiente código crea un array llamado `unArray`:

```
using System;
class RecorriendoArrays
{
    public static void Main()
    {
        int[] unArray = {4, 5, 6, -1, 0};
        foreach (int i in unArray)
        {
            System.Console.WriteLine(i);
        }
    }
}
```

La salida de este código es:

```
4
5
6
-1
0
```

El siguiente programa recorre un array y describe cada uno de sus elementos como par o impar.

```
using System;
class RecorriendoArrays
{
    public static void Main()
    {
        int[] unArray = {4, 5, 6, -1, 27};
        int par=0;
        int impar=0;
        string dos="par";
        string uno="impar";
        foreach(int i in unArray)
        {
```



```
        Console.WriteLine(i);
        if(i%2==0)
        {
            par++;
            Console.WriteLine("El elemento {0} es {1}",i,dos);
        }

        else
        {
            impar++;
            Console.WriteLine("El elemento {0} es {1}",i,uno);
        }
    }
    Console.WriteLine("Hay {0} pares y {1} impares" ,par, impar);
}
}
```

La salida de este programa es:

```
4
El elemento 4 es par
5
El elemento 5 es impar
6
El elemento 6 es par
-1
El elemento -1 es impar
27
El elemento 27 es impar
Hay dos pares y 3 impares
```

Colecciones

Introducción

Anteriormente se ha estudiado cómo utilizar los arrays para referenciar un conjunto de objetos o de variables. Es evidente, sin embargo que los arrays tienen algunas limitaciones. La mayor de ella es que una vez que el programador ha creado el array, su tamaño no se puede cambiar porque es fijo, constante. El problema se presenta cuando se pretende añadir nuevos elementos al array, sin crear uno nuevo.

En general, se puede decir que una colección se utiliza para trabajar con listas o conjuntos ordenados de objetos y proporciona una funcionalidad mayor que la de un simple array. Esta funcionalidad proviene de la implementación de una serie de interfaces del namespace `System.Collections`. Este namespace también contiene clases que implementan estos interfaces y facilitan enormemente la tarea del programador.

Las colecciones proporcionan métodos básicos para acceder a los elementos de la colección utilizando corchetes, de manera idéntica a como se accede a los elementos de un array.

C# proporciona una serie de clases e interfaces que están contenidas en el namespace `System.Collections`, que nos permite trabajar conjuntos de datos o colecciones, de manera muy sencilla.

Los interfaces que proporcionan funcionalidad a las colecciones, son los siguientes:

- `IEnumerable`: Proporciona la capacidad para “recorrer” una colección a través de sus elementos, por medio de una sentencia `foreach`.

- `ICollection`: Hereda de `IEnumerable`. Proporciona la capacidad para obtener el número de elementos de la colección y de copiar elementos a un simple array.

- `IList`: Hereda de `IEnumerable` y de `ICollection`. Proporciona una lista de los elementos de la colección con las capacidades de los interfaces anteriormente citados y algunas otras capacidades básicas.

- `IDictionary`: Hereda de `IEnumerable` y de `ICollection`. Proporciona una lista de elementos de la colección accesibles a través de un valor en lugar de un índice.

Los arrays, en C# son objetos de la clase `System.Array` que es un tipo de colección. La clase `System.Array` hereda de `IList`, `ICollection` e `IEnumerable` pero no proporciona algunas funcionalidades avanzadas de `IList`. Representa una lista de elementos con un tamaño fijo. El programador puede también crear sus propias colecciones específicas.

La clase `ArrayList`

Una de las clases más importantes que proporciona el namespace `System.Collections` se denomina `System.Collections.ArrayList`, que implementa las interfaces `IList`, `ICollection` e `IEnumerable`.

Este tipo puede utilizarse para representar una lista de elementos con un tamaño variable, es decir es un array cuyo tamaño puede cambiar dinámicamente cuando sea necesario. Proporciona un determinado número de métodos y propiedades para manipular sus elementos.

Algunos de los más importantes son los siguientes:

<code>Adapter()</code>	Método estático que crea un <code>ArrayList</code> para un objeto que implementa <code>IList</code> .
<code>Capacity</code>	Determina o lee el número de elementos del <code>ArrayList</code> .
<code>Count</code>	El número actual de elementos del <code>ArrayList</code> .
<code>Item()</code>	Obtiene o fija el elemento correspondiente a su índice determinado.
<code>Add()</code>	Añade elementos al <code>ArrayList</code> .
<code>AddRange()</code>	Permite añadir los elementos de una <code>ICollection</code> al final del <code>ArrayList</code> .

Clear ()	Elimina todos los elementos del ArrayList.
Contains ()	Determina si un elemento está en la ArrayList.
Index Of ()	Devuelve el índice de un determinado elemento.
Insert ()	Inserta un elemento en un ArrayList.
InsertRange()	Inserta elementos de una colección en un ArrayList.
Remove	Elimina un determinado elemento.
RemoveAt	Elimina un determinado elemento accediendo a él a través de su índice.
Sort()	Ordena un ArrayList.
ToArray()	Copia los elementos del ArrayList a un array.

El siguiente ejemplo ilustra bien la diferencia entre un Array y un ArrayList puede ser:

Considere la clase `Animal` almacenada en el fichero `Animal.cs` con el siguiente código:

```
using System;
namespace ConsoleApplication2
{
    public class Animal
    {
        protected string nombre;
        public Animal(string unNombre)
        {
            nombre=unNombre;
        }
        public string Nombre
        {
            get
            {
                return nombre;
            }
            set
            {
                nombre=value;
            }
        }
        public void Comer()
        {
            Console.WriteLine(" el animal {0} ha comido",nombre);
        }
    }
}
```

A continuación, modifique el código en el método `Main()` por el siguiente:

```
using System;
namespace ConsoleApplication2
{
    class MiAplicacion
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Ejemplo de Array****");
            //Se define un array de dos elementos de la clase Animal
            Animal[] arrayAnimal = new Animal[2];
        }
    }
}
```

```

        Animal gato=new Animal ("Mimoso");
        arrayAnimal[0]=gato;
        arrayAnimal[1]=new Animal("Andrajoso");

        // se recorre el array por medio de la sentencia foreach
        Console.WriteLine("\nRecorremos el array con
                           foreach");
        foreach (Animal an in arrayAnimal)
            Console.WriteLine(an.Nombre);

        // se recorre el array por medio de la sentencia for
        Console.WriteLine("\nRecorremos el array con una
                           sentencia for");
        for (int i=0;i<arrayAnimal.Length;i++)
            Console.WriteLine(arrayAnimal[i].Nombre);

        Console.WriteLine("El número de elementos del array es
        {0}",arrayAnimal.Length);
    }
}

```

La salida de este programa es:

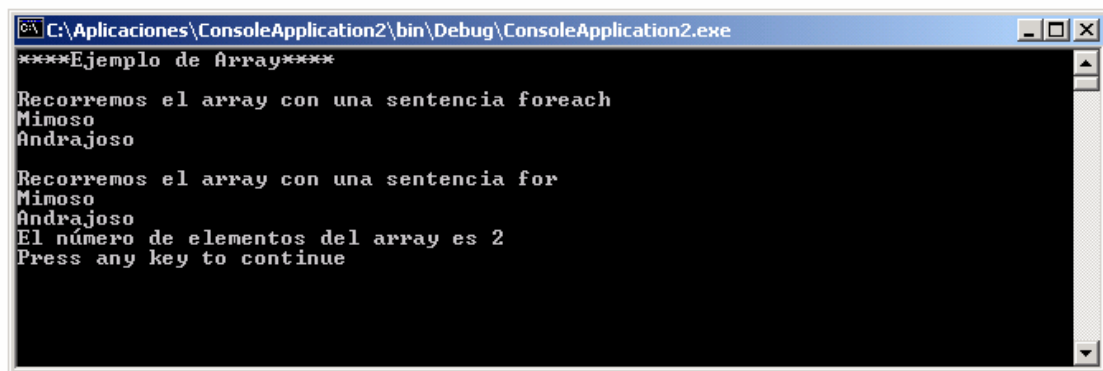


Figura 7.1

En este ejemplo se observa bien la diferencia que existe cuando se accede a los elementos del array por medio de las sentencias for o foreach

En el siguiente ejemplo, se pretende trabajar de forma parecida con una colección de tipo `ArrayList`. Para ello, además de añadir la directiva:

```
using System.Collections;
```

añada el siguiente código al final del método `Main()`:

```

Console.WriteLine("\n***Ejemplo de ArrayList*****");

ArrayList listaArrayAnimales = new ArrayList();
listaArrayAnimales.Add(new Animal("Perico"));
listaArrayAnimales.Add(new Animal("Anquilosado"));
Animal pantera=new Animal("Mougli");
listaArrayAnimales.Add (pantera);

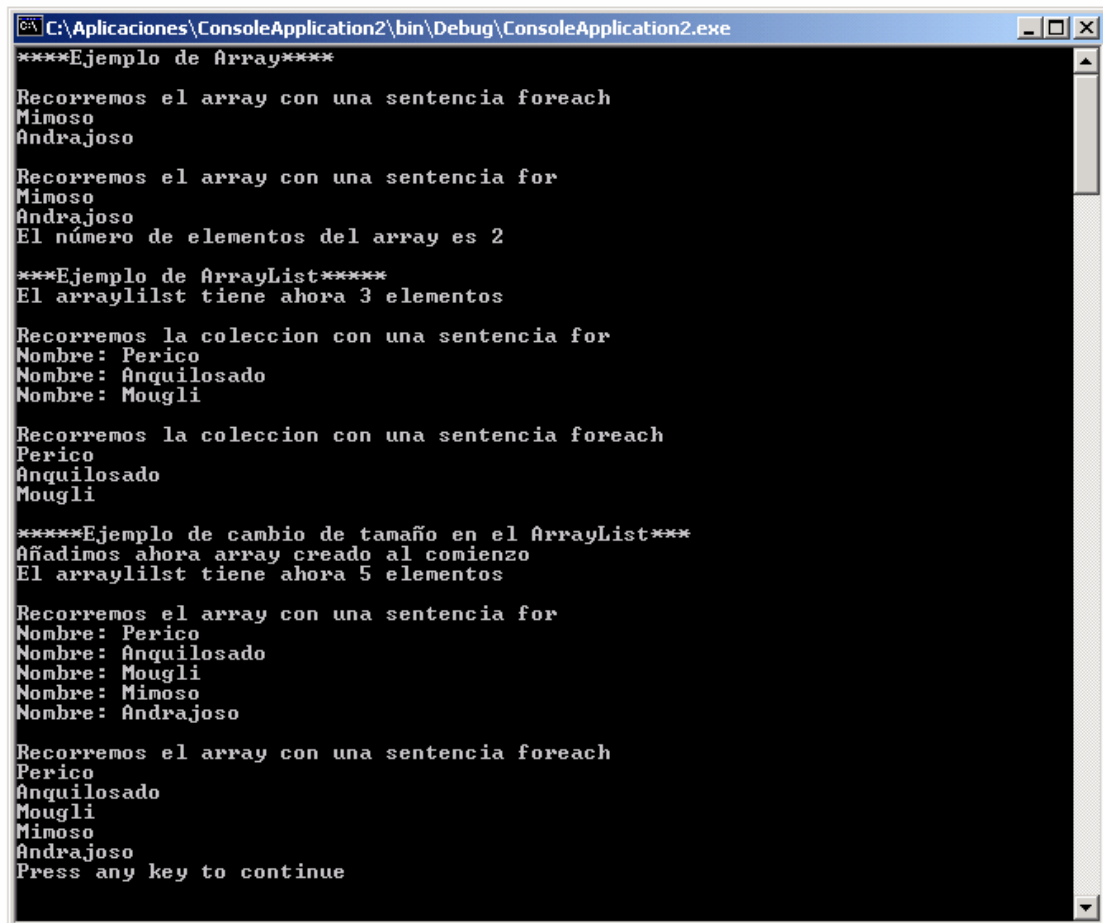
```

```

Console.WriteLine("El arraylilst tiene ahora {0} elementos",
    listaArrayAnimales.Count);
//Puede utilizarse unArrayList como un Array
//Los elementos de un ArrayList son objetos,y es necesario
//convertirlos a objetos de la clase Animal al utilizarlo como array
Console.WriteLine("\nRecorremos la coleccion con una sentencia for");
for(int i=0;i<listaArrayAnimales.Count;i++)
    Console.WriteLine("Nombre: {0} ",
        ((Animal)listaArrayAnimales[i]).Nombre);
Console.WriteLine("\nRecorremos la coleccion con una sentencia
    foreach");
//Aquí no es necesaria la conversión
foreach(Animal an in listaArrayAnimales)
    Console.WriteLine(an.Nombre);
//Un ArrayList puede cambiar de tamaño
Console.WriteLine("\n*****Ejemplo de cambio de tamaño en el
    ArrayList***");
Console.WriteLine("Añadimos ahora array creado al comienzo");
listaArrayAnimales.AddRange(arrayAnimal);
Console.WriteLine("El arraylilst tiene ahora {0} elementos",
    listaArrayAnimales.Count);
Console.WriteLine("\nRecorremos el array con una sentencia for");
for(int i=0;i<listaArrayAnimales.Count;i++)
    Console.WriteLine("Nombre: {0} ",
        ((Animal)listaArrayAnimales[i]).Nombre);
Console.WriteLine("\nRecorremos el array con una sentencia foreach");
foreach(Animal an in listaArrayAnimales)
    Console.WriteLine(an.Nombre);

```

La salida de este programa es:



```

C:\Aplicaciones\ConsoleApplication2\bin\Debug\ConsoleApplication2.exe
****Ejemplo de Array****
Recorremos el array con una sentencia foreach
Mimoso
Andrajoso

Recorremos el array con una sentencia for
Mimoso
Andrajoso
El número de elementos del array es 2

****Ejemplo de ArrayList****
El arraylilst tiene ahora 3 elementos

Recorremos la coleccion con una sentencia for
Nombre: Perico
Nombre: Anquilosado
Nombre: Mougli

Recorremos la coleccion con una sentencia foreach
Perico
Anquilosado
Mougli

****Ejemplo de cambio de tamaño en el ArrayList***
Añadimos ahora array creado al comienzo
El arraylilst tiene ahora 5 elementos

Recorremos el array con una sentencia for
Nombre: Perico
Nombre: Anquilosado
Nombre: Mougli
Nombre: Mimoso
Nombre: Andrajoso

Recorremos el array con una sentencia foreach
Perico
Anquilosado
Mougli
Mimoso
Andrajoso
Press any key to continue

```

Figura 7.2

En el anterior ejemplo se han creado dos colecciones de objetos: el primero utiliza la clase `System.Array` –que de hecho es una colección–, y el segundo utiliza la clase `System.Collections.ArrayList`. Ambas colecciones tienen objetos de la clase `Animal` que está definida en el fichero `Animal.cs`. Respecto a la manipulación del array, probablemente el lector encontrará pocas dificultades para su comprensión. Sin embargo, es importante señalar aquí que cuando se crea un array se especifica un tamaño, que será fijo y constante a lo largo del programa:

```
Animal[] arrayAnimal = new Animal[2];
```

Cuando se crea un objeto de la clase `ArrayList`, no se define ningún tamaño. Para crear la colección, simplemente se utiliza el código siguiente:

```
ArrayList listaArrayAnimales = new ArrayList();
```

Existen otros dos constructores de esta clase. El primero de ellos, utiliza como parámetro una colección y el otro, especifica la capacidad –la propiedad `capacity`– o número de elementos inicial de la colección y se pasa como parámetro entero.

Inicialmente, la colección `ArrayList` no tiene ningún elemento. No se puede añadir objetos a través del índice, como en los arrays. Para ello, se utiliza el método `Add()` como se ve en el ejemplo:

```
listaArrayAnimales.Add(new Animal("Perico"));
listaArrayAnimales.Add(new Animal("Anquilosado"));
Animal pantera=new Animal("Mougli");
listaArrayAnimales.Add (pantera);
```

Una vez que se ha añadido un elemento, se puede modificar como si fuera un array, aunque no se ha hecho anteriormente. Por ejemplo:

```
listaArrayAnimales[0]=new Animal("Pepe");
```

Se puede recorrer, por medio de una sentencia `foreach`, tanto un array como un `ArrayList` porque ambos implementan la interface `IEnumerable`:

```
foreach(Animal an in arrayAnimal)
    Console.WriteLine(an.Nombre);

foreach(Animal an in listaArrayAnimales)
    Console.WriteLine(an.Nombre);
```

Se puede acceder al número de elemento o tamaño de un array por medio de la propiedad `Length`:

```
Console.WriteLine("El número de elementos del array es {0}",
    arrayAnimal.Length);
```

Y al tamaño de una colección `ArrayList` por medio de la propiedad `Count`, gracias a que esta clase implementa la interface `ICollection`:

```
Console.WriteLine("El arraylist tiene ahora {0} elementos",
    listaArrayAnimales.Count);
```

La diferencia más importante es que un array está fuertemente tipado, y permite acceder a sus elementos directamente:

```
Console.WriteLine(arrayAnimal[i].Nombre);
```

Cuando se accede a los elementos por medio del índice debe realizarse una conversión de tipos:

```
Console.WriteLine("Nombre: {0} ",
    ((Animal)listaArrayAnimales[i]).Nombre);
```

Finalmente, pueden eliminarse objetos de una colección por medio de los métodos

```
Remove(object unObjeto) O RemoveAt(int indice).
```

Por ejemplo, si se pretende eliminar el elemento `gato` y el primer elemento de la colección, han de añadirse estas líneas al final del código del método `Main()`:

```
listaArrayAnimales.Remove(gato);
listaArrayAnimales.RemoveAt(0);
Console.WriteLine("El arraylist tiene ahora {0} elementos",
    listaArrayAnimales.Count);
```

Además puede obtenerse el índice de un determinado elemento por medio del método `IndexOf()`, por ejemplo:

```
Console.WriteLine(listaArrayAnimales.IndexOf(gato));
```

Si no existe el elemento en la colección, este método devuelve `-1`.

Además, puede añadirse una colección a un objeto de la clase `ArrayList`, por medio del método `AddRange()`, al que se le pasa la colección como parámetro. Dicha colección puede ser también un array, que es una colección también. Por ejemplo, en la siguiente línea se añade el array `arrayAnimal` al `ArrayList`:

```
listaArrayAnimales.AddRange(arrayAnimal);
```

Las colecciones son imprescindibles para trabajar con algunos controles como `ListBox`, `ComboBox`, etc, y su conocimiento reduce enormemente el esfuerzo de la programación de este tipo de controles.