



C#

Arreglos y Colecciones

Ing. Juan Carlos Méndez N.



ARREGLOS

Arreglos

- Un arreglo es el tipo de colección más simple, soportado directamente por el lenguaje
- Se trata de un secuencia de objetos del mismo tipo que pueden ser accedidos por medio de un índice
- Cuando se declara un arreglo, realmente se está instanciando un objeto de una sub-clase de `System.Array`, que implementa `ICloneable`, `IList`, `ICollection` e `IEnumerable`
- Se cuenta con Lo mejor de dos mundos: una sintaxis familiar para declarar arreglos y la definición de una clase que brinda acceso a un conjunto de métodos y propiedades comunes a todos los arreglos
- Se pueden efectuar conversiones de tipo entre arreglos, siempre que sus dimensiones sean iguales y sea posible convertir (implícita o explícitamente) entre los tipos de sus elementos

Arreglos - Declaración

`[[atributos]] [modificadores] tipo[] nombre;`

- Los `[]` indican al compilador que se está declarando un arreglo, y se los conoce como el operador índice.
- El tipo indica el tipo de elementos que puede contener el arreglo
- Se instancian usando la palabra clave `new`
- Los arreglos son tipos referencia y se crean en el *heap*. Los elementos del arreglo se crean en la pila o en el *heap*, según se traten de tipos valor o tipos referencia respectivamente

Arreglos - Inicialización y Acceso

- En el momento de instanciar un arreglo sus elementos quedan inicializados en los valores por defecto correspondientes a su tipo

```
string[] miArreglo;  
miArreglo = new string[10];
```

- Pueden ser inicializados explícitamente en el momento de su declaración:

```
int[] miArreglo = new int[5] { 2, 4, 6, 8, 10 };  
int[] miArreglo = { 2, 4, 6, 8, 10 };
```

- El índice de los arreglos empieza en 0 y los elementos se acceden usando la sintaxis `nombre[i]`, donde *i* es el elemento ubicado en la posición *i*+1 dentro del arreglo
- Si se trata de acceder un elemento por fuera del rango de un arreglo, la excepción `IndexOutOfRangeException` será lanzada

Clase System.Array

Método o Propiedad	Descripción
IsFixedSize	Propiedad que indica si el arreglo es de tamaño fijo
Length	Propiedad que retorna el tamaño del arreglo
Rank	Propiedad que retorna el número de dimensiones del arreglo
Clear()	Establece el valor de un rango de elementos de un arreglo en 0 o null
Copy()	Copia una sección de un arreglo a otro arreglo
IndexOf()	Retorna el índice de la primera ocurrencia de un valor en un arreglo unidimensional
LastIndexOf()	Retorna el índice de la última ocurrencia de un valor en un arreglo unidimensional
Reverse()	Invierte el orden de los elementos en un arreglo unidimensional
Sort()	Ordena los elementos (que deben implementar <code>IComparable</code>) de un arreglo unidimensional usando el algoritmo <i>Quicksort</i>

Array - Propiedad Length

- Indica cuántos objetos hay en un arreglo, retornando el *tamaño total* de éste
- En el caso de un arreglo multidimensional, retorna el tamaño total de todo el arreglo *no* el tamaño de su “primer nivel”

```
1. using System;
2. public class TestLength {
3.     int[,] MyArray = new int[2,5];
4.     public static void Main() {
5.         TestLength c = new TestLength();
6.         Console.WriteLine(c.MyArray.Length);
7.         Console.WriteLine(c.MyArray.GetLength(0));
8.         Console.WriteLine(c.MyArray.GetLength(1));
9.     }
10. }
```

Array - Ejemplo

```
1.    using System;
2.
3.    public class Tester {
4.
5.        static void Main() {
6.
7.            String[] myArray = {"Who", "is", "John", "Doe"};
8.
9.            PrintMyArray(myArray);
10.           Array.Reverse(myArray);
11.           PrintMyArray(myArray);
12.
13.           String[] myOtherArray =
14.               {"We", "Hold", "These", "Truths", "To", "Be", "Self", "Evident"};
15.
16.           PrintMyArray(myOtherArray);
17.           Array.Sort(myOtherArray);
18.           PrintMyArray(myOtherArray);
19.           Console.ReadLine();
20.       }
21.
22.       public static void PrintMyArray(object[] theArray) {
23.           foreach (object obj in theArray)
24.               Console.WriteLine("Value: {0}", obj);
25.           Console.WriteLine();
26.       }
27.   }
```


Age Group	Percentage
18-24	85%
25-34	75%
35-44	65%
45-54	55%
55-64	45%
65-74	35%
75-84	25%
85+	10%



Age Group	Percentage
18-24	85%
25-34	75%
35-44	65%
45-54	55%
55-64	45%
65-74	35%
75-84	25%
85+	10%

Age Group	Percentage
18-24	85%
25-34	75%
35-44	65%
45-54	55%
55-64	45%
65-74	35%
75-84	25%
85+	10%

Arreglos Rectangulares

`[[atributos]] [modificadores] tipo[,] nombre;`

- Un arreglo rectangular es un arreglo de dos o más dimensiones. En uno de dos dimensiones, la primera es el numero de filas y la segunda el número de columnas
- El número de “,” en la declaración indica el número de dimensiones del arreglo
- Los elementos se acceden usando la sintaxis `nombre[i, j]`

Arreglos Rectangulares - Ejemplo

```
1.    using System;
2.
3.    class Tester {
4.
5.        public void Run() {
6.
7.            const int rows = 4;
8.            const int columns = 3;
9.
10.           // declara un arreglo rectangular 4x3 de enteros
11.           int[,] rectangularArray = { {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11} };
12.
13.           // imprime el contenido del arreglo
14.           for (int i = 0; i < rows; i++)
15.               for (int j = 0; j < columns; j++)
16.                   Console.WriteLine("[{0},{1}] = {2}", i, j, rectangularArray[i,j]);
17.
18.        }
```

```
13.    static void Main() {
14.        Tester t = new Tester();
15.        t.Run();
16.        Console.ReadLine();
17.    }
18. }
```

Arreglos Serrados

`[[atributos]] [modificadores] tipo[][] nombre;`

- Cada fila puede tener un número diferente de columnas
- Un arreglo serrado puede entenderse como un arreglo en donde cada uno de sus elementos es un arreglo
- Al momento de declararlos, se debe indicar el número de filas
- El número de “[]” en la declaración indica el número de dimensiones del arreglo
- También pueden inicializarse en la declaración:

```
int[][] miArreglo = new int[2][]  
    { new int[] {1,2,3}, new int[] {4,5,6,7,8} };
```
- Los elementos se acceden usando la sintaxis `nombre[i][j]`

Arreglos Serrados - Ejemplo

```
1.     using System;
2.
3.     class Tester {
4.
5.         public void Run() {
6.
7.             const int rows=4, rowZero=5, rowOne=2, rowTwo=3, rowThree=5;
8.
9.             // declara un arreglo serrado de cuatro filas
10.            int[][] jaggedArray = new int[rows][];
11.
12.            // declara cada fila con tamaños distintos
13.            jaggedArray[0] = new int[rowZero];
14.            jaggedArray[1] = new int[rowOne];
15.            jaggedArray[2] = new int[rowTwo];
16.            jaggedArray[3] = new int[rowThree];
17.
18.            // llena algunas (pero no todas las) posiciones
19.            jaggedArray[0][3] = 15;
20.            jaggedArray[1][1] = 12;
21.            jaggedArray[2][1] = 9;
22.            jaggedArray[2][2] = 99;
23.            jaggedArray[3][0] = 10;
24.            jaggedArray[3][1] = 11;
25.            jaggedArray[3][2] = 12;
26.            jaggedArray[3][3] = 13;
27.            jaggedArray[3][4] = 14;
28.
29.            for(int i=0;i<rowZero;i++) Console.WriteLine("[0][{0}] = {1}",i,jaggedArray[0][i]);
30.            for(int i=0;i<rowOne;i++) Console.WriteLine("[1][{0}] = {1}",i,jaggedArray[1][i]);
31.            for(int i=0;i<rowTwo;i++) Console.WriteLine("[2][{0}] = {1}",i,jaggedArray[2][i]);
32.            for(int i=0;i<rowThree;i++) Console.WriteLine("[3][{0}] = {1}",i,jaggedArray[3][i]);
33.
34.        }
35.
36.        static void Main() {
37.            new Tester().Run();
38.            Console.ReadLine();
39.        }
40.    }
```



COLECCIONES

Colecciones

- Una colección es un contenedor que alberga un grupo de objetos
- El *framework* .NET proporciona varias clases que implementan la funcionalidad de una colección
- Igualmente, proporciona un conjunto de interfaces estándar para enumerar, comparar y crear colecciones
- Se aconseja estudiar el espacio de nombres `System.Collections` para profundizar en el tema de las colecciones

Interfaces Útiles

Interfaz	Métodos
ICollection	<code>int Count {get;}</code> <code>bool IsSynchronized {get;}</code> <code>object SyncRoot {get;}</code> <code>void CopyTo(Array array, int index)</code>
IComparer	<code>int Compare(object x, object y)</code>
IEnumerable	<code>IEnumerator GetEnumerator()</code>
IEnumerator	<code>object Current {get;}</code> <code>bool MoveNext()</code> <code>void Reset()</code>
IDictionaryEnumerator	<code>DictionaryEntry Entry {get;}</code> <code>object Key {get;}</code> <code>object Value {get;}</code>
IHashCodeProvider	<code>int GetHashCode(object obj)</code>

Interfaces Útiles

Interfaz	Métodos
ICollection	<code>bool IsFixedSize {get;}</code> <code>bool IsReadOnly {get;}</code> <code>object this[int index] {get; set;}</code> <code>int Add(object value)</code> <code>void Clear()</code> <code>bool Contains(object value)</code> <code>int IndexOf(object value)</code> <code>void Insert(int index, object value)</code> <code>void Remove(object value)</code> <code>void RemoveAt(int index)</code>
IDictionary	<code>bool IsFixedSize {get;}</code> <code>bool IsReadOnly {get;}</code> <code>object this[object key] {get; set;}</code> <code>ICollection Keys {get;}</code> <code>ICollection Values {get;}</code> <code>void Add(object key, object value)</code> <code>void Clear();</code> <code>bool Contains(object key)</code> <code>IDictionaryEnumerator GetEnumerator();</code> <code>void Remove(object key)</code>

IComparer e IComparable

- La interfaz `IComparer` proporciona el método `Compare()` por medio del cual dos objetos cualquiera en una colección pueden ser ordenados
- Típicamente, `IComparer` es implementado en clases auxiliares, cuyas instancias son pasadas como parámetros a métodos como `Array.Sort(Array a, IComparer c)`
- El *framework* .NET provee las clases `Comparer` y `CaseInsensitiveComparer` que implementan `IComparer` y permiten efectuar comparaciones de cadenas de caracteres - teniendo o no en cuenta mayúsculas/minúsculas, respectivamente-
- La interfaz `IComparable` es similar, pero define `CompareTo()` en el mismo objeto que va a ser comparado y no en una clase auxiliar
- La ventaja de tener una clase auxiliar implementando `IComparer` es que permite tener un mayor control sobre la forma en que se efectúa el ordenamiento. Por ej., es posible definir varios criterios distintos para ordenar una colección de objetos del mismo tipo

IEnumerable e IEnumerator

- Las instancias de colecciones que implementan IEnumerable pueden ser recorridas fácilmente usando foreach
- Además de una clase que implemente IEnumerable, se necesita otra clase auxiliar que implemente IEnumerator
- La responsabilidad del enumerador será la de proporcionar los métodos necesarios para recorrer una colección: Current, MoveNext() y Reset()

IEnumerable e IEnumerator - Ejemplo

```
1.    using System;
2.    using System.Collections;

3.    class TestForeach {
4.
5.        static void Main() {
6.
7.            string s = "abc123";
8.            Console.WriteLine(s is IEnumerable + "\n");
9.
10.           foreach (char c in s) Console.WriteLine(c);
11.
12.           Console.WriteLine("\n---\n");
13.
14.           IEnumerator ie = s.GetEnumerator();
15.           while (ie.MoveNext()) {
16.               char c = (char) ie.Current;
17.               Console.WriteLine(c);
18.           }
19.
20.           Console.ReadLine();
21.
22.       }
23.
24.   }
```

IDictionary e IDictionaryEnumerator

- Las clases que implementan `IDictionary` permiten representar una colección que asocia un valor a una determinada llave y se las conoce como Diccionarios
- Dichas clases también pueden ser enumeradas usando `foreach`, pero con una diferencia: el método `GetEnumerator()` retorna un objeto que implementa `IDictionaryEnumerator`
- Un Diccionario puede verse como un conjunto de objetos que encapsulan una pareja llave-valor
- Los objetos que implementen `IDictionaryEnumerator` son consistentes con esta idea: un diccionario es enumerado recorriendo instancias de la estructura `DictionaryEntry`, que encapsulan una llave y su respectivo valor

Clase `System.Collections.ArrayList`

Método o Propiedad	Descripción
<code>Capacity</code>	Propiedad que indica el número de elementos que actualmente puede contener el <code>ArrayList</code>
<code>Count</code>	Propiedad que retorna el número de elementos en la colección
<code>Item</code>	Indexador de la clase <code>ArrayList</code>
<code>Add()</code>	Añade un objeto al final
<code>Clear()</code>	Remueve todos los elementos
<code>GetEnumerator()</code>	Retorna el enumerador
<code>Insert()</code>	Inserta un elemento en un índice dado
<code>RemoveAt()</code>	Remueve un elemento de un índice dado
<code>Reverse()</code>	Invierte el orden de los elementos
<code>Sort()</code>	Ordena los elementos usando <i>Quicksort</i>
<code>ToArray()</code>	Copia todos los elementos a un nuevo arreglo

ArrayList - Ejemplo

```
1.    using System;
2.    using System.Collections;

3.    public class Employee {
4.
5.        private int empID;
6.
7.        public Employee(int empID) {
8.            this.empID = empID;
9.        }
10.
11.        public override string ToString() {
12.            return empID.ToString();
13.        }
14.    }
15.
16.    public class Tester {
17.
18.        static void Main() {
19.
20.            ArrayList empArray = new ArrayList();
21.            ArrayList intArray = new ArrayList();
22.
23.            // Llenamos los arreglos
24.            for (int i = 0; i < 5; i++) empArray.Add(new Employee(i+100));
25.            for (int i = 0; i < 17; i++) intArray.Add(i);
26.
27.            // Imprimimos el contenido del ArrayList de enteros
28.            for (int i = 0; i < intArray.Count; i++) Console.Write("{0} ", intArray[i]);
29.            Console.WriteLine("\n\nintArray.Capacity: {0}", intArray.Capacity);
30.
31.            // Imprimimos el contenido del ArrayList de empleados
32.            for (int i = 0; i < empArray.Count; i++) Console.Write("{0} ", empArray[i]);
33.            Console.WriteLine("\n\nempArray.Capacity: {0}", empArray.Capacity);
34.            Console.ReadLine();
35.        }
36.    }
37. }
```

Clase System.Collections.Queue

Método o Propiedad	Descripción
Count	Propiedad que retorna el número de elementos en la colección
Clear()	Remueve todos los elementos
Contains()	Determina si un elemento está en la cola
CopyTo()	Copia un rango de elementos a un arreglo ya existente
Dequeue()	Remueve y retorna el objeto al comienzo de la cola
Enqueue()	Añade un objeto al final de la cola
GetEnumerator()	Retorna el enumerador
Peek()	Retorna el objeto al comienzo de la cola sin removerlo
ToArray()	Copia todos los elementos a un nuevo arreglo

Queue - Ejemplo

```
1.    using System;
2.    using System.Collections;

3.    public class Tester {
4.
5.        static void Main() {
6.
7.            Queue intQueue = new Queue();
8.
9.            // Pone elementos en la cola
10.           for (int i = 0; i < 5; i++) intQueue.Enqueue(i*5);
11.           PrintValues(intQueue);

12.           // Remueve un elemento de la cola
13.           Console.WriteLine("\n(Dequeue)\t{0}", intQueue.Dequeue());
14.           PrintValues(intQueue);

15.           // Mira el primer elemento de la cola sin removerlo
16.           Console.WriteLine("\n(Peek) \t{0}", intQueue.Peek());
17.           PrintValues(intQueue);
18.           Console.ReadLine();
19.
20.       }

21.       public static void PrintValues(IEnumerable myCollection) {
22.           Console.Write("intQueue values:\t");
23.           foreach (object o in myCollection) Console.Write("{0} ", o);
24.           Console.WriteLine();
25.       }
26.   }
```

Clase System.Collections.Stack

Método o Propiedad	Descripción
Count	Propiedad que retorna el número de elementos en la colección
Clear()	Remueve todos los elementos
Contains()	Determina si un elemento está en la pila
CopyTo()	Copia un rango de elementos a un arreglo ya existente
GetEnumerator()	Retorna el enumerador
Peek()	Retorna el objeto en la parte superior de la pila sin removerlo
Pop()	Remueve y retorna el objeto en la parte superior de la pila
Push()	Añade un objeto en la parte superior de la pila
ToArray()	Copia todos los elementos a un nuevo arreglo

Stack - Ejemplo

```
1.     using System;
2.     using System.Collections;

3.     class Tester {

4.         public static void Main() {

5.             Stack intStack = new Stack();

6.             // Pone elementos en la pila
7.             for (int i = 0; i < 8; i++) intStack.Push(i*5);
8.             PrintValues(intStack);

9.             // Remueve un elemento de la pila
10.            Console.WriteLine("\n(Pop)\t{0}", intStack.Pop());
11.            PrintValues(intStack);

12.            // Mira el primer elemento de la pila sin removerlo
13.            Console.WriteLine("\n(Peek) \t{0}", intStack.Peek());
14.            PrintValues(intStack);
15.            Console.ReadLine();

16.        }

17.        public static void PrintValues(IEnumerable myCollection) {
18.            Console.Write("intStack values:\t");
19.            foreach (object o in myCollection) Console.Write("{0} ", o);
20.            Console.WriteLine();
21.        }

22.    }
```

Clase System.Collections.Hashtable

Método o Propiedad	Descripción
Count	Propiedad que retorna el número de elementos en la colección
Keys	Retorna una ICollection con las llaves de la Hashtable
Values	Retorna una ICollection con los valores de la Hashtable
Item	Indexador de la clase Hashtable
Add()	Añade una entrada con la llave y el valor especificados
Clear()	Remueve todos los elementos
ContainsKey()	Determina si la Hashtable contiene una llave especificada
ContainsValue()	Determina si la Hashtable contiene un valor especificado
CopyTo()	Copia un rango de elementos a un arreglo ya existente
GetEnumerator()	Retorna el enumerador
Remove()	Remueve una entrada con la llave especificada

Hashtab1e

- Un tipo de diccionario optimizado para recuperar rápidamente los valores
- Tanto llaves como valores pueden ser cualquier object o tipo básico
- Por defecto, se usa el método GetHashCode() como función de *hashing* y el método Equals() para determinar si dos objetos son iguales, ambos heredados de object
- También se puede implementar la interfaz IHashCodeProvider para definir una nueva función de *hashing*
- CaseInsensitiveHashCodeProvider implementa dicha interfaz para calcular un *hash* de cadenas de caracteres que no tenga en cuenta mayúsculas/minúsculas
- Recordar que se pueden probar distintos valores de capacidad y factor de carga para acomodar la Hashtab1e a la colección de elementos que se quieren almacenar y lograr un buen balance entre: minimizar colisiones, maximizar el uso eficiente de la memoria y acceder rápidamente los valores almacenados

Hashtable - Ejemplo

```
1.     using System;
2.     using System.Collections;

3.     public class TestClass {

4.         public static void Main() {

5.             Hashtable ht = new Hashtable();

6.             // Usando Add()
7.             ht.Add("A", "apple"); ht.Add("D", "durian");
8.             ht.Add("B", "banana"); ht.Add("C", "coconut");
9.             PrintCollection(ht);
10.            // Usando la propiedad Count
11.            Console.WriteLine("Count: {0}\n", ht.Count);
12.            // Usando ContainsKey()
13.            Console.WriteLine(ht.ContainsKey("D"));
14.            Console.WriteLine(ht.ContainsKey("S") + "\n");
15.            // Usando ContainsValue()
16.            Console.WriteLine(ht.ContainsValue("banana"));
17.            Console.WriteLine(ht.ContainsValue("starfruit") + "\n");
18.            // Usando Remove()
19.            ht.Remove("B");
20.            PrintCollection(ht);
21.            // Usando la propiedad Keys
22.            foreach (string key in ht.Keys) Console.WriteLine(key);
23.            // Usando la propiedad Values()
24.            Console.WriteLine();
25.            foreach (string val in ht.Values) Console.WriteLine(val);
26.            Console.ReadLine();

27.        }

28.        public static void PrintCollection(IEnumerable myCollection) {
29.            foreach (DictionaryEntry de in myCollection)
30.                Console.WriteLine("Key: {0} Value: {1}", de.Key, de.Value);
31.            Console.WriteLine();
32.        }

33.    }
```

Colecciones - Usos Avanzados

- Todas las clases que implementan `ICollection` tienen métodos para trabajar con versiones sincronizadas de éstas
- `BitArray` es una colección similar a `ArrayList`, diseñada para trabajar con bits representados como valores booleanos
- `SortedList` es un híbrido entre `Hashtable` y `Array`, almacena llaves-valores, pero sus llaves están ordenadas. Sus valores pueden recuperarse usando llaves o índices
- Estudiar el espacio de nombres `System.Collections.Specialized`. Cuenta con colecciones especializadas y optimizadas para trabajar con un tipo de datos determinado, por ej., cadenas de texto

**Muchas
Gracias**