Colecciones en C#.

En muchas ocasiones es necesario agrupar los datos de una aplicación en contenedores compuestos (“Colecciones”), para que dicho conjunto de datos pueda ser tratado como una única entidad en lugar de una serie de elementos aislados.

Para ejemplificar, podría citarse el caso de una aplicación que administre los alumnos que participan de una clase.

En tal caso, y a partir de la implementación de una clase "CAlumno" que modele las características y comportamientos de los alumnos que participan del curso, deberían luego constituirse tantas instancias de la clase "CAlumno" como estudiantes estén matriculados.

```
CAlumno a1 = new CAlumno(“Juan”);
CAlumno a2 = new CAlumno(“Pedro”);
CAlumno a3 = new CAlumno(“Luisa”);
…
CAlumno a35 = new CAlumno(“Marcos”);
```

Una colección de datos, es una clase que nos permite agrupar un conjunto de objetos en uno solo, brindándonos la posibilidad de agregar, consultar, modificar y eliminar objetos de la colección.

Manejar en este caso 35 alumnos cada uno por separado, obviamente no nos resultaría muy práctico, ya que no nos permitiría automatizar los procesos de acceso a cada una de las instancias, nos dificultaría ofrecerla como argumento de un método, entre otros ejemplos.

La solución estándar a esta problemática, es el empleo de **colecciones**, entidades que nos permitirán agrupar un conjunto de elementos, gestionándolos (accediendo, consultando, agregando, eliminando) inherentemente como tal.

Colecciones básicas: Las Matrices.

Más allá de algunas diferencias sintácticas y muy pocas funcionales, en prácticamente todos los lenguajes de programación existen colecciones básicas de datos.

El ejemplo de colección más elemental son las ***Matrices*** (cuyo caso más común es la matriz unidimensional o ***vector***), también llamadas ***Arreglos*** (del inglés *“Array”*).

Las matrices, consisten colecciones dimensionadas estáticamente (una vez establecido su tamaño, no podrán redimensionarse) cuyos elementos son todos de la misma naturaleza o tipo.

En el ámbito del Framework .Net, las matrices son instancias de una clase particular: la clase ***Array***, del espacio de nombres ***System*** (`System.Array`).

Matrices Unidimensionales (Vectores).

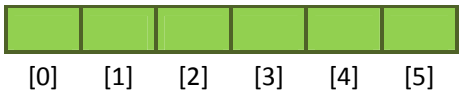

Figura 1: Representación de una matriz unidimensional (vector).

Son aquellos arreglos organizados en una única dimensión. Su forma general de instanciación para matrices unidimensionales es la siguiente:

```
Ámbito tipo[] identificador;
identificador = new tipo[cantidad_elementos];
```

O bien, simultáneamente:

```
Ámbito tipo[] identificador = new tipo[cantidad_elementos];
```

Por ejemplo si queremos crear un vector de 10 elementos (índice 0 a 9) cada uno de tipo entero estándar (en C#: **int**), procederemos:

```
int [] vector;
vector = new int[10];
```

o bien:

```
int [] vector = new int[10];
```

Cuando los elementos de una colección explicitan un tipo (como es en el caso de las matrices) y ése tipo corresponde a uno de los tipos de datos ***primitivos*** (denominados así por su homogeneidad funcional con los tipos de datos básicos de los lenguajes estructurados de programación: *int*, *float*, *bool*, *double*, etc.), cada uno de los elementos de la colección se instancian automáticamente, asumiendo el valor por defecto del tipo.

Así, en el caso del ejemplo, para cada elemento del vector, se crea una instancia del tipo *int*, asumiendo cada una de ellas el valor por defecto del tipo: 0 (cero).

Cuando tiene lugar el caso de colecciones de objetos en general, como en nuestro ejemplo del curso y sus 35 alumnos, la creación de la correspondiente matriz de alumnos, podría instrumentarse:

```
CAlumno [] listado;
listado = new CAlumno[10];
```

o bien:

```
CAlumno listado = new CAlumno[35];
```

Hemos así creado una instancia de la clase Array.

Ahora, dado que los elementos de la matriz no son del tipo primitivo, no se instanciarán automáticamente y por ello, cada uno de ellos referenciará (“apuntará”) a una instancia aún inexistente (***null***).

Por ello, será menester, crear una instancia por cada elemento de la colección y hacerla apuntar por éste. Ello podría realizarse:

```
CAlumno auxAlumno;
auxAlumno = new CAlumno(“Juan”);
listado[0]=auxAlumno;
auxAlumno = new CAlumno(“Pedro”);
listado[1]=auxAlumno;
auxAlumno = new CAlumno(“Luisa”);
listado[2]=auxAlumno;
…
auxAlumno = new CAlumno(“Marcos”);
listado[34]=auxAlumno;
```

o, directamente:

```
listado[0] = new CAlumno("Juan");
listado[1] = new CAlumno("Pedro");
listado[2] = new CAlumno("Luisa");
…
listado[34] = new CAlumno("Marcos");
```

Matrices Multidimensionales.

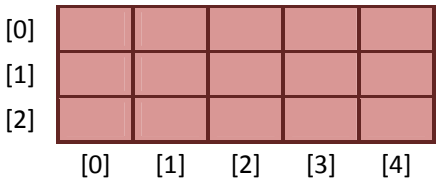

Figura 2: Representación de una matriz bidimensional.

Cuando se trata de matrices multidimensionales: tablas (2 dimensiones), cubos (3 dimensiones) o poliedros (4 o más dimensiones), su forma general de instanciación es:

```
Ámbito tipo[,,…] identificador;
identificador = new tipo[elem_1°Dimensión,elem_2°Dimensión,… ];
```

Nótese que la cantidad de operadores, ("coma") más uno, es igual a la cantidad de dimensiones del arreglo.

Por ejemplo si queremos crear una matriz tridimensional de 10 x 3 x 2 elementos de tipo flotante de precisión simple (en C#: **float**), procederemos:

```
float [,,] cubo;
cubo = new float[10,3,2];
```

o bien:

```
float [,,] cubo = new float[10,3,2];
```

Para instanciar una matriz bidimensional de objetos de 5 x 4 elementos *CAlumno,* puede hacerse:

```
CAlumno [,] tabla;
tabla = new CAlumno[5,4];
```

o bien:

```
CAlumno tabla = new CAlumno[5,4];
```

Tengamos presente que hemos creado una instancia de la clase *Array*, debiendo luego, instanciar cada uno de los elementos *CAlumno* de la matriz. Es decir:

```
tabla[0,0] = new CAlumno("Juan");
```

```
tabla[0,1] = new CAlumno("Pedro");
…
tabla[2,1] = new CAlumno("Andrea");
…
tabla[4,3] = new CAlumno("Julieta");
```

Matrices Irregulares

Se denominan ***matrices irregulares***, ***dentadas*** ("***jagged***" en inglés), o ***arreglo de arreglos***, a aquellas en las cuales, recorriéndolas en el sentido de una o más dimensiones, encontramos diferentes cantidades de elementos.

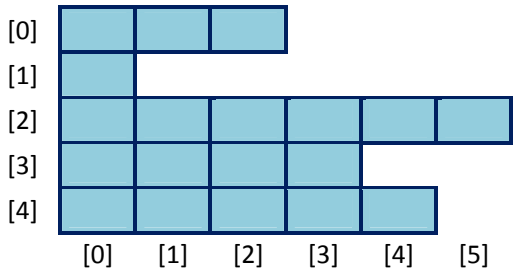

Figura 3: Representación de una matriz bidimensional Irregular.

Se las denomina también *Arreglo de Arreglos,* pues pueden concebirse como una matriz, cuyos elementos son, a su vez, también matrices.

Su forma sintáctica general de creación es:

```
Ámbito tipo[][]...[] identificador;
identificador = new tipo[n][];// n: cantidad de elementos.
identificador [0]= new tipo[j][];// j: cantidad de elementos.
identificador [1]= new tipo[k][];// k: cantidad de elementos.
…
identificador [n-1]= new tipo[m][];// m: cantidad de elementos.
```

Ahora, en instanciación de matrices irregulares, encontraremos un par de corchetes por cada dimensión del arreglo.

Por ejemplo si queremos crear una matriz bidimensional como la de la Figura 3, con elementos de tipo lógico (en C#: **bool**):

```
bool[][] jagg;
jagg = new bool[5][];
jagg[0] = new bool[3];
jagg[1] = new bool[1];
jagg[2] = new bool[6];
jagg[3] = new bool[4];
jagg[4] = new bool[5];
```

Del mismo modo podremos proceder al instanciar matrices dentadas de elementos tipo objeto, cuidando de instanciar luego, cada uno de ellos en la clase correspondiente.

Métodos y Propiedades de la Clase *Array*

Acceso (Lectura y Escritura):

Dado que empleamos metodologías inherentes a la programación orientada a objetos, para el acceso a los elementos de un vector, instancia de la clase *Array*, emplearemos para el acceso, en lugar de funciones, métodos.

En este caso, serán los métodos de instancia SetValue () y GetValue (), para la escritura y lectura respectivamente.

arreglo.SetValue (valor, índices_elemento)
arreglo.GetValue (índices_elemento)

Ejemplo

Escritura del valor 500 en el 4º elemento del vector numeros:

```
numeros.SetValue(500,3);
```

Lectura del valor presente en el 2º elemento del vector numeros y asignación sobre una variable:

```
variableDestino = numeros.GetValue (1);
```

Determinación de límites de un arreglo:

En el Framework .Net existe una fuerte comprobación de límites, inhibiéndose el acceder fuera de los límites del arreglo (esto puede incluso lanzar una excepción en tiempo de ejecución).

Para evitar esa contingencia, disponemos de los métodos de instancia GetLowerBound () y GetUpperBound (), para determinar respectivamente los límites inferior y superior del mismo, debiéndose especificar como argumento, la dimensión cuyos límites se desea determinar.

arreglo.GetUpperBound (Dimensión) As Integer
arreglo.GetLowerBound (Dimensión) As Integer
(Si se trata de un vector, Dimensión=0)

Ejemplo

```
limSup = numeros.GetUpperBound (0);
limInf = numeros.GetLowerBound (0);
```

Determinación de la cantidad de elementos de un arreglo:

Aunque la cantidad de elementos de un arreglo es perfectamente determinable a partir de sus límites y dimensiones, disponemos de la propiedad de instancia Length (“Longitud”) y del método de instancia GetLength () para obtener la extensión de elementos de un objeto de la clase *Array*.

arreglo.Length
arreglo.GetLength (Dimensión)

Ejemplo

```
cantElem = numeros.Length;
cantElem = numeros.GetLength(0);
```

Importante: si se emplea la propiedad Length, para determinar la cantidad de elementos de un arreglo multidimensional, se obtendrá la cantidad total de elementos.

Ordenamiento de arreglos.

El ordenamiento de los valores almacenados en los elementos de un arreglo, es uno de los procesos más habituales y suele ser indispensable, como precedente de otros, tales como la búsqueda binaria.

Por tal motivo, el Framework .Net propone un método de clase (no de instancia), Sort (), que permite el ordenamiento de menor a mayor, de una colección de datos almacenada en un arreglo.

Dado que se trata de un método de clase, su invocación se efectúa, no sobre una instancia de la clase, sino sobre la clase misma, la clase *Array*.

Array.Sort (arreglo, [posición_inicial, cantidad_de_elementos_a_ordenar])[1]

Ejemplo

```
Array.Sort(numeros, 2, 4);
Array.Sort(numeros);
```

Búsqueda en arreglos.

Hallar el índice del elemento donde se almacena un valor a buscar, puede implementarse simplemente haciendo uso del método de clase BinarySearch (), que además, ha de hacerlo empleando el más eficiente de los métodos de búsqueda: la búsqueda binaria.

Ahora si, deberá tenerse presente, que dicha búsqueda binaria exige que el arreglo se halle ordenado y por ello, debemos ordenar previamente al arreglo de menor a mayor, si no se hallare naturalmente ordenado.

Dado que se trata de un método de clase, su invocación se efectúa, no sobre una instancia de la clase, sino también, sobre la propia clase *Array*.

Array.BinarySearch (arreglo, [posic_inicial, cant_elementos], valor_a_buscar)

Ejemplo

```
indice = Array.BinarySearch(numeros, 5);
```

Inversión de orden en arreglos.

La inversión de orden de los valores almacenados en los elementos de un arreglo, implica, simplemente, el "reflejo" del valor presente en el primer elemento, en el último y viceversa; del segundo elemento en el anteúltimo y viceversa; así sucesivamente hasta alcanzar el intercambio de posición a extremos de todos los elementos del arreglo. Para ello, hace uso del método de clase Reverse().

Representa un proceso indispensable para obtener un ordenamiento de mayor a menor, combinados con el método Sort(), aunque esta no es, obviamente, su único campo de aplicación.

Array.Reverse (arreglo, [posición_inicial, cantidad_de_elementos_a_ordenar])

Ejemplo

```
Array.Reverse(numeros, 1, 5);
Array.Reverse(numeros);
```

Búsqueda lineal en vectores.

El método de búsqueda binaria es óptimo cuando, sobre un arreglo se han de efectuar varios procesos de búsqueda.

[1] Recordemos: Los corchetes denotan opcionalidad.

Sin embargo, en aquellos casos donde la búsqueda sea única, el costo del ordenamiento del arreglo invalida la optimización propuesta por la búsqueda binaria.

Hallar el índice del elemento donde se almacena un valor a buscar, puede implementarse simplemente haciendo uso de los métodos de clase IndexOf ( ) y LastIndexOf ( ) que desarrollan una búsqueda lineal, comenzando, respectivamente, desde el primer elemento del arreglo hacia el último, en un caso, y en el segundo, en sentido inverso.

Opcionalmente, puede especificarse un elemento inicial, distinto de los elementos-extremo de la colección.

variable = Array.IndexOf (vector, valor_a_buscar, [posición_inicial_de_búqueda])
variable = Array.LastIndexOf (vector, valor_a_buscar, [posición_inicial_de_búqueda])

Ejemplo

```
indice = Array.IndexOf(numeros, 5);
```

Copia y duplicación de arreglos.

Cuando por algún motivo debidamente fundado, se desea duplicar o "clonar" un arreglo, o bien copiar total o parcialmente los elementos de un arreglo en otro, de modo tal que las modificaciones efectuadas sobre el segundo, no afecten al arreglo original primero, como sucedería en el caso de una simple referencia múltiple, puede hacerse uso de los métodos CopyTo () (de instancia), Copy () (de clase) y Clone () (de instancia) de la clase *Array*.

Los dos primeros, permiten copiar total o parcialmente, los elementos de un arreglo "origen" hacia un arreglo "destino", y exigen conocer "a priori", la naturaleza del arreglo a copiar, pues dicha copia se efectúa siempre, hacia un arreglo de idéntico tipo.

Además, la longitud del arreglo destino, en CopyTo () deberá coincidir estrictamente con la cantidad de elementos a copiar más los elementos no utilizados del arreglo destino, mientras que en Copy () sólo es necesario que sea de longitud igual o mayor.

El tercero, carece de tal exigencia, pues duplicar o "clonar" implica dos procesos: crear y copia.

Conforme a esto, el arreglo destino se instanciará ("creará") a partir de la invocación del método sobre una instancia dada de la clase *Array*.

Por ello, la referencia sobre la que se instanciará dicho arreglo destino, deberá ser, conforme la flexibilidad propuesta por la herencia de clases, del tipo definido en la superclase de cualquier arreglo o matriz: la clase *Array*.

Array.Copy(origen, posic_inic_origen, destino, posic_inic_destino , cant_elementos)
origen.CopyTo (destino, posición_inicial_destino)
referenciaNoInstanciada = origen.Clone ( )

Ejemplo

```
numeros.CopyTo(copiaA,0);
Array.Copy(numeros,0,copiaA,0,6);
Array duplicado;
duplicado = numeros.Clone();
```