**COLECCIONES EN C SHARP**

Una de las tareas más comunes que se presentan en la ejecución de una aplicación, es la de **agrupar datos en colecciones** para que puedan ser tratadas como un *único objeto* en lugar de tratar cada uno por separado.

Un ejemplo claro podría ser una aplicación que administre los alumnos que se tienen en un aula de clase.

En ese caso, se tendría que ***crear una clase "Alumno"*** que represente en nuestra aplicación a cada uno de los alumnos que hacen parte del curso. Luego, en algún momento, tendremos la necesidad de crear tantas instancias de la clase "Alumno" como estudiantes estén matriculados.

Alumno a1=new Alumno();   
Alumno a2=new Alumno();   
Alumno a3=new Alumno(); .   
.   
Alumno a35=new Alumno();

**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 no es muy práctico ya que se necesitarían muchas líneas de código para realzar alguna operación sobre todos los alumnos.

Supongamos que en la aplicación se da la posibilidad de marcar la asistencia o no asistencia de un alumno a la clase. Adicionalmente tenemos un método que recibirá como parámetro la lista entera de los alumnos y los marcara a todos como si hubieran asistido a clase (Algo práctico para el encargado de registrar dicha información).

Si tratamos a cada alumno por separado, tendríamos los dos siguientes problemas:

* Tendríamos que codificar una lista enorme de parámetros para el método (Un total de 35 parámetros, todos de tipo "Alumno"). Esto sería bastante tedioso de codificar y luego de leer.

El método sería algo como esto.

public void marcarAsistencia(Alumno a1, Alumno a2, ………,Alumno a35)   
{   
}

* Si por algún motivo el número de estudiantes varia, tendríamos que modificar el código fuente para que el método reciba el numero de parámetros correcto, lo cual haría nuestra aplicación poco escalable (evolucionable).

Por otro lado, si tratamos a nuestros alumnos como una colección de datos, tendremos la ventaja de que siempre el método recibirá una colección (independiente del número de alumnos, siempre será una colección). 

En todos los lenguajes de programación existen las colecciones básicas de datos, la diferencia entre un lenguaje y otro es la sintaxis que se debe utilizar para implementarla.

La *colección más básica* que se puede utilizar es el **Array (Arreglo o Vector**).

Consiste en un objeto con *tamaño fijo* al que podemos agregar otros objetos. En general, la sintaxis a utilizar es la siguiente.

Tipo de dato [] identificador=new Tipo de dato [longitud fija];

Por ejemplo si queremos crear un vector de enteros de 10 posiciones:

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

Para el caso especifico de nuestro ejemplo, tendríamos que crear un vector de Alumnos para almacenar un total de 35 alumnos.

Alumno listaAlumnos=new Alumno[35];

Luego de crear el vector de alumnos, procedemos a llenar la colección con los datos que contendrá. Los vectores utilizan índices para tener acceso a los objetos que contiene. Hay que tener presente que los índices de los vectores son basados en cero, es decir, que siempre el primer elemento tendrá un índice de 0 y no de 1.

Así, si queremos almacenar en la primera posición del vector el alumno a1, haríamos lo siguiente.

listaAlumno[0]=a1;

Y así sucesivamente.

listaAlumno[0]=a1;    
listaAlumno[1]=a2;    
listaAlumno[2]=a3;    
.   
.   
.   
listaAlumno[34]=a35;

De esta manera, el método que se encarga de marcar que todos los alumnos asistieron a clase, quedaría así:

public void marcarAsistencia(Alumno[] lista)   
{   
}

La verdad es que el método nos a quedado mucho mas legible, sin embargo, la escalabilidad de nuestro código sigue siendo pobre porque si el numero de alumnos varia toca modificar la sentencia de creación del vector y su respectiva asignación del valor.

Por esta razón, existen diferentes tipos de colecciones que permiten un mejor manejo de datos según sea necesario en nuestra aplicación.

**ARRAYLIST**

**Esta es una clase que representa una lista de datos**.

El *ArrayList* puede **aumentar o disminuir su tamaño dinámicamente** de una manera eficiente.

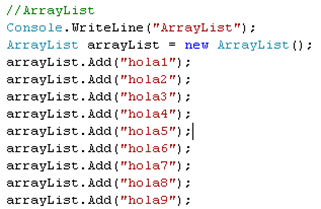
Con un array de datos no era posible aumentar la capacidad del vector ya que dicho parámetro es especificado en el momento de crear la instancia del objeto. El ArrayList a diferencia, brinda la posibilidad de aumentar o disminuir su tamaño dinámicamente según sea necesario.

Para crear una instancia de este objeto, se debe utilizar la clase ArrayList incluida en el espacio de nombre ***System.Collections*** como se muestra a continuación.

**ArrayList NombreArrayList=new ArrayList();**

El constructor de la clase ArrayList acepta también un parámetro tipo entero que indica la capacidad inicial del objeto que se está creando.

Si es necesario agregar un objeto a la colección, se debe utilizar el **método Add**, el cual inserta el nuevo elemento en la última posición, o el **método Insert** el cual lo inserta en la posición indicada.

[](https://developmania.files.wordpress.com/2008/08/image8.png)

Todos los objetos almacenados en un Arraylist ***son tratados como objetos***, por lo tanto, es posible agregar todo tipo de datos, es decir, se puede agregar enteros, cadenas de texto, objetos de clases propias, etc. Y a ***diferencia de los array***, no todos los elementos deben ser del mismo tipo de dato.

Esto en algunas ocasiones puede ser una ventaja ya que permite almacenar gran variedad de información en una sola colección, sin embargo, por razones de rendimiento (cast, boxing, unboxing), hay ocasiones en las que es preferible utilizar las colecciones genéricas que serán tratadas mas adelante.

Si es necesario ***quitar elementos de la colección***, se debe usar el método **remove, removeAt o RemoveRange**, los cuales eliminan el objeto pasado como parámetro, o un elemento en una posición especifica, o un grupo de elementos respectivamente.

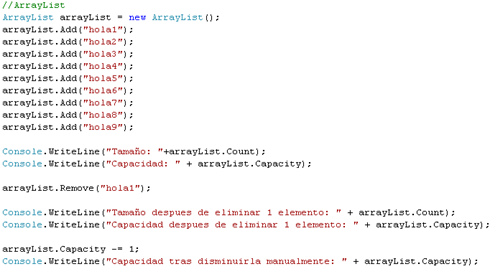
Las propiedades más utilizadas de esta colección son: **Count y Capacity**.

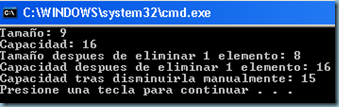
La primera sirve para conocer la cantidad actual de elementos que contiene la colección.

**La segunda** indica la capacidad máxima actual de la colección para almacenar elementos. Es necesario tener presente que la capacidad de la colección, aumenta en caso de ser necesario al insertar un elemento, con lo que se garantiza el redimensionamiento automático.

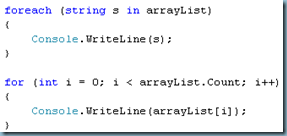
La capacidad de una colección nunca podrá ser menor a la cantidad total de elementos contenidos, por lo que si se modifica manualmente la propiedad Capacity y se le asigna un valor menor que el valor devuelto por la propiedad Count, obtendremos una excepción de tipo ArgumentOutOfRangeExcepetion .

A continuación se muestra un ejemplo del uso de estas propiedades:

[](https://developmania.files.wordpress.com/2008/08/image9.png)

[](https://developmania.files.wordpress.com/2008/08/image10.png)

Para acceder a los elementos contenidos por la colección se puede hacer mediante el uso de índices o mediante la ***instrucción foreach***.

[](https://developmania.files.wordpress.com/2008/08/image11.png)

Determinar qué tipo de colección usar en un caso especifico, es tarea del desarrollado y se debe evaluar las condiciones para determinar la manera más eficiente de administrar los recursos.

Si es un escenario donde no conocemos el tamaño que tendrá la colección y si además será muy probable que el tamaño varíe, entonces será recomendable bajo todas las demás circunstancias usar un ArrayList en lugar de un array debido a que el ArrayList brinda la posibilidad de redimensionarlo automáticamente.

Sin embargo, para escenarios donde se conoce de antemano la cantidad total de elementos a almacenar y si todos son del mismo tipo, se debe usar el array convencional ya que los objetos son almacenados en su tipo de datos nativo y no es necesario hacer conversiones.