

## Colecciones de datos

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 realizar 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 marcará 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 varía, tendríamos que modificar el código fuente para que el método reciba el número 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 específico 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 queda mucho más legible, sin embargo, la escalabilidad de nuestro código sigue siendo pobre porque si el número 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

## Vectores

Como lo explique colecciones de datos son algunas de las herramientas más utilizadas a la hora de desarrollar una aplicación ya que nos permiten manipular información en memoria de una manera fácil y flexible.

La plataforma .NET ofrece una gran variedad de colecciones que debemos conocer y saber utilizar para emplear la más indicada en nuestras aplicaciones y obtener así el mejor rendimiento posible tanto en procesamiento como en manejo de memoria. A continuación se explican algunas de las colecciones incluidas en el lenguaje C#.NET.

La clase Array, contenida en el namespace System, provee toda la funcionalidad necesaria en las colecciones para manipular los datos. Esta es una clase abstracta, por lo cual no es posible crear instancias de la misma, sino que sirve como clase base (de la que otras clases heredan) para las demás clases que representan colecciones en .NET

La colección mas sencilla que utiliza como clase base System.Array, es lo que se conoce comúnmente como vector. Para definir un vector, debemos tener claro que tipo de dato queremos que nuestro vector almacene y la cantidad de valores que deseamos almacenar ya que son requisitos indispensables para la definición y creación de vectores.

En C#, un vector se define con un par de corchetes [] después del tipo de dato, como se muestra a continuación.

```
string[] texto;  
bool[] booleanos;
```

Sin embargo, para hacer uso de nuestro vector, debemos primero crear la instancia declarando cuantas posiciones máximas tendrá, como se muestra a continuación:

```
booleanos = new bool[3];  
texto = new string[100];
```

Luego de esto, podremos almacenar tantos datos del mismo tipo como posiciones tenga el vector. Por ejemplo en la variable booleanos se podrá almacenar hasta 3 variables de tipo bool y en la variable texto se podrá almacenar hasta 100 variables de tipo string. Es importante tener en cuenta que la primera posición de un vector es la posición 0, por lo cual si la longitud es de 5, las posiciones van desde la 0 hasta la 4.

Para almacenar información en un vector, solo debemos hacer referencia a la posición del vector donde deseamos almacenarla y utilizar el operador de asignación (=).

```
texto[0] = "Texto 1";  
texto[1] = "Texto 2";  
texto[2] = "Texto 3";  
texto[3] = "Texto 4";
```

En el ejemplo anterior, se han almacenado 4 cadenas de texto (strings) en el vector texto declarado e instanciado anteriormente. Cada string se ha almacenado en una posición del vector diferente. En caso de almacenar un string en una posición donde ya se había almacenado anteriormente algo, el valor antiguo será sobrescrito por el nuevo valor.

A continuación se almacena en cada posición restante del vector una cadena de texto mediante un operador de control de flujo (for). Este operador es posible utilizarlo con todas las colecciones en .NET ya que todas son enumerables debido a que implementan la interface IEnumerable.

```
for (int i = 4; i < texto.Length; i++)  
{  
    texto[i] = "Texto " + (i + 1);  
}  
  
for (int i = 0; i < texto.Length; i++)  
{  
    Console.WriteLine("Posicion " + i + " : " + texto[i]);  
}
```

Un vector puede tener mas de una dimensión según sea necesario. Un vector de dos dimensiones es considerado una matriz de datos.

Para definir un vector de mas de una dimensión, se utiliza el carácter ',' en los corchetes de la definición del mismo.

```
int [] a = new int[2]; //Una dimensión
```

```
int [ , ] a = new int [2,3]; //Dos dimensiones
```

```
int [ , , ]a = new int[2,3,4]; //Tres dimensiones
```

Las propiedades mas importantes que exponen los vectores se presentan a continuación:

**Length:** Indica la longitud total del vector (En todas las dimensiones).

**Rank:** Indica el numero total de dimensiones del vector.

Para obtener la longitud total de un determinada dimensión, se debe utilizar el método **GetLength** que recibe como parámetro un entero indicando el índice de la dimensión de la cual se desea conocer la longitud (La primera dimensión tiene como índice 0).

Aunque es posible crear vectores de mas de una dimensión, no es recomendable utilizar mucho esta técnica en las aplicaciones ya que la complejidad de nuestro algoritmo se incrementa considerablemente, teniendo en cuenta que el numero de iteraciones necesarias para recorrer todas las posiciones del vector aumenta de manera exponencial en función del numero de dimensiones. Generalmente el numero máximo de dimensiones que se utiliza en un vector es 2 cuando se necesita trabajar con matrices de datos.

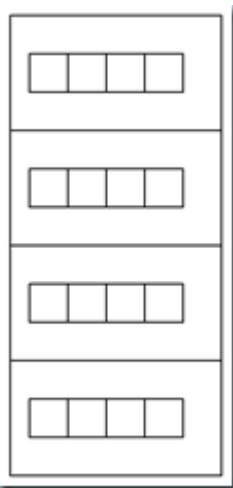
A continuación se muestra un ejemplo de una matriz de datos:

```
//Matriz de 2 filas y 3 columnas
string[,] matriz=new string[2,3];

//Mostrando la longitud en cada dimension
for (int i = 0; i < matriz.Rank; i++)
{
    Console.WriteLine("Longitud de la dimension "+i+" : "+
        matriz.GetLength(i));
}

//recorriendo todas las posiciones
for (int i = 0; i < matriz.GetLength(0); i++)
{
    for (int j = 0; j < matriz.GetLength(1); j++)
    {
        matriz[i,j] = (i + j).ToString();
        Console.WriteLine(matriz[i, j]);
    }
}
```

Otra de las colecciones que se pueden crear con los vectores es un jagged array o en español un array de arrays. Se puede pensar en un jagged array como un vector de una dimensión, donde en cada posición en lugar de almacenar un tipo de dato específico, almacena otro vector.



En esta imagen se ilustra gráficamente como se vería un jagged array. En este ejemplo, se tiene un vector exterior de 4 posiciones y en cada una de las posiciones almacena un vector también de 4 posiciones. No es obligación que todos los vectores sean de la misma longitud.

Para declarar un array de arrays en C#, se debe utilizar la siguiente sintaxis:

```
int [][] jagged;
```

Para instanciar un jagged array se debe indicar el numero de posiciones que tendrá el vector contenedor, como se hace con un vector normal de una dimensión.

```
int [][] jagged = new int[3][];
```

En el ejemplo anterior se ha definido un array de tres posiciones, donde se almacenara en cada una de ellas, un array de enteros (indicado por el segundo par de corchetes). Luego de inicializar el vector contenedor, podremos crear los vectores internos como se muestra a continuación:

```
//array de arrays
int [][] jagged = new int[3][];

//inicializar cada uno de los arrays interiores
jagged[0] = new int[5];
jagged[1] = new int[3];
jagged[2] = new int[15];

//Accediendo a posiciones especificas
jagged[0][4] = 3;
```

Igualmente se puede crear un array de arrays de varias dimensiones, donde cada vector almacenado tendrá 2, 3 o mas dimensiones según sea necesario.

```
//Definiendo un jagged array de dos dimensiones
int[,] jagged2 = new int[3][,];

jagged2[0] = new int[2,3];
jagged2[1] = new int[5,6];
jagged2[2] = new int[8,8];
```

Sin embargo, la complejidad en este caso aumenta bastante y se debe buscar siempre obtener el mejor rendimiento posible en nuestras aplicaciones, por lo tanto estas colecciones complejas deben ser utilizadas únicamente cuando sea necesario y cuando ninguna de las colecciones disponibles en la plataforma .NET es suficiente para almacenar la información.

## Array List

Esta es una clase que representa una lista de datos. Es bastante parecida a la colección explicada [anteriormente](#), con la diferencia que 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 arrayList=new ArrayList();
```

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

```
//ArrayList
Console.WriteLine("ArrayList");
ArrayList arrayList = new ArrayList();
arrayList.Add("hola1");
arrayList.Add("hola2");
arrayList.Add("hola3");
arrayList.Add("hola4");
arrayList.Add("hola5");
arrayList.Add("hola6");
arrayList.Add("hola7");
arrayList.Add("hola8");
arrayList.Add("hola9");
```

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 vistos en la parte I, 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 específica, o un grupo de elementos respectivamente.

Las propiedades mas 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 ArgumentOutOfRangeException .

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

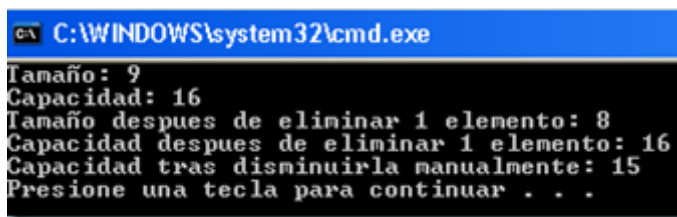
```
//ArrayList
ArrayList arrayList = new ArrayList();
arrayList.Add("hola1");
arrayList.Add("hola2");
arrayList.Add("hola3");
arrayList.Add("hola4");
arrayList.Add("hola5");
arrayList.Add("hola6");
arrayList.Add("hola7");
arrayList.Add("hola8");
arrayList.Add("hola9");

Console.WriteLine("Tamaño: "+arrayList.Count);
Console.WriteLine("Capacidad: " + arrayList.Capacity);

arrayList.Remove("hola1");

Console.WriteLine("Tamaño despues de eliminar 1 elemento: " + arrayList.Count);
Console.WriteLine("Capacidad despues de eliminar 1 elemento: " + arrayList.Capacity);

arrayList.Capacity -= 1;
Console.WriteLine("Capacidad tras disminuirla manualmente: " + arrayList.Capacity);
```



```
C:\WINDOWS\system32\cmd.exe
Tamaño: 9
Capacidad: 16
Tamaño despues de eliminar 1 elemento: 8
Capacidad despues de eliminar 1 elemento: 16
Capacidad tras disminuirla manualmente: 15
Presione una tecla para continuar . . .
```

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

```
foreach (string s in arrayList)
{
    Console.WriteLine(s);
}

for (int i = 0; i < arrayList.Count; i++)
{
    Console.WriteLine(arrayList[i]);
}
```

Determinar que tipo de colección usar en un caso específico, es tarea del desarrollador 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.



# Stack y Queue

Las pilas (Stack) y las colas (Queue) son dos colecciones muy similares entre si ya que solo varia la forma en que guardan y extraen los elementos que contienen. En ciertas cosas estas dos colecciones se parece a un ArrayList, como por ejemplo que soporta el redimensionamiento automático y que los elementos son almacenados como objetos (System.Object). Pero también tienen algunas diferencias, como por ejemplo que no se puede cambiar su capacidad y no se puede acceder a sus elementos a través de índices

En algunas ocasiones, es importante tener un control sobre el orden en que los elementos son ingresados y obtenidos de la colección. Por esta razón existen las colecciones Stack y Queue. Como se menciona anteriormente, en estas colecciones NO es posible acceder aleatoriamente mediante índices a sus elementos, sino que es necesario utilizar un método encargado de extraer un elemento a la vez. Pero cual elemento?. Precisamente en la respuesta a esa pregunta radica la diferencia entre estas dos colecciones.

La pila (Stack), es una colección en la que todo nuevo elemento se ingresa al final de la misma, y únicamente es posible extraer el ultimo elemento de la colección. Por este comportamiento, el Stack es conocido como una colección LIFO (Last Input Fisrt Output) ya que siempre el ultimo elemento ingresado a la colección, será el primero en salir. Quizás la mejor manera de recordar el comportamiento de un Stack, es asociándolo con una “pila” de platos en donde cada plato esta encima del otro y en caso de querer ingresar un plato a la pila, lo que se debe hacer es ponerlo encima del ultimo plato. Luego cuando se quiere sacar un plato de la pila, solo podemos coger el ultimo plato.

La cola (Queue), tiene el comportamiento contrario a la pila. Todo nuevo elemento se agrega al principio de la colección y solo se puede extraer el ultimo elemento. Por esta razón, la cola se conoce como una colección FIFO (Fisrt Input First Output) ya que el primer elemento que ingresa a la cola es el primer elemento que sale. Para recordar este comportamiento se puede asociar la Queue con la fila que se debe hacer en un banco para realizar una consignación. En ese caso, el cajero atiende en el orden en que llegan las personas a la cola.

Las colecciones Stack y Queue se encuentran en el espacio de nombres System.Collections como todas las colecciones no genéricas.

Para implementar cada una de ellas se debe utilizar la clase Stack y Queue respectivamente y utilizar sus métodos que ofrecen la posibilidad de agregar elementos a la colección y extraer elementos según el comportamiento de la colección que se este utilizando.

```
Stack stack = new Stack();  
stack.Push(2);  
stack.Push(3);
```

```
Queue queue = new Queue();  
queue.Enqueue(4);  
queue.Enqueue(5);
```

Como se ve en la figura anterior, para agregar elementos a una pila se debe utilizar el método Push que recibe como parámetro un Object. Mientras que en la cola se debe utilizar el método Enqueue (encolar). Ambos métodos, incrementan automáticamente la capacidad de la colección.

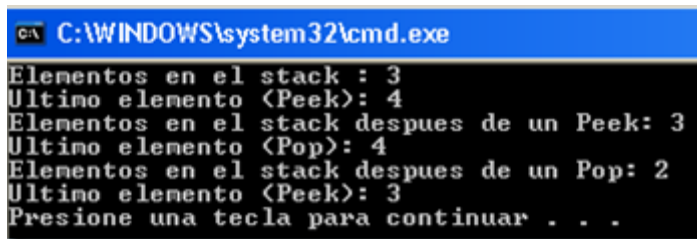
Para obtener un elemento de la colección, contamos con dos opciones diferentes:

1. Obtener el elemento indicado según el comportamiento de la colección sin quitarlo de la colección. Esto se logra mediante el método Peek de cada colección.
2. Obtener un elemento de la colección, quitándolo de la misma. Esto se logra mediante el método Pop de la pila (Stack) o el método Dequeue de la cola (Queue).

Ejemplo del Stack:

```
Stack stack = new Stack();
stack.Push(2);
stack.Push(3);
stack.Push(4);
Console.WriteLine("Elementos en el stack : "+stack.Count);
Console.WriteLine("Ultimo elemento (Peek): "+stack.Peek());
Console.WriteLine("Elementos en el stack despues de un Peek: "+stack.Count);
Console.WriteLine("Ultimo elemento (Pop): "+stack.Pop());
Console.WriteLine("Elementos en el stack despues de un Pop: "+stack.Count);
Console.WriteLine("Ultimo elemento (Peek): " + stack.Peek());
```

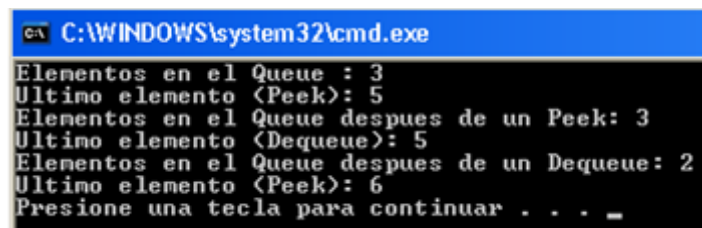
Resultado:



```
C:\WINDOWS\system32\cmd.exe
Elementos en el stack : 3
Ultimo elemento <Peek>: 4
Elementos en el stack despues de un Peek: 3
Ultimo elemento <Pop>: 4
Elementos en el stack despues de un Pop: 2
Ultimo elemento <Peek>: 3
Presione una tecla para continuar . . .
```

Ejemplo del Queue:

```
Queue queue = new Queue();
queue.Enqueue(5);
queue.Enqueue(6);
queue.Enqueue(7);
Console.WriteLine("Elementos en el Queue : " + queue.Count);
Console.WriteLine("Ultimo elemento (Peek): " + queue.Peek());
Console.WriteLine("Elementos en el Queue despues de un Peek: " + queue.Count);
Console.WriteLine("Ultimo elemento (Dequeue): " + queue.Dequeue());
Console.WriteLine("Elementos en el Queue despues de un Dequeue: " + queue.Count);
Console.WriteLine("Ultimo elemento (Peek): " + queue.Peek());
```



```
C:\WINDOWS\system32\cmd.exe
Elementos en el Queue : 3
Ultimo elemento <Peek>: 5
Elementos en el Queue despues de un Peek: 3
Ultimo elemento <Dequeue>: 5
Elementos en el Queue despues de un Dequeue: 2
Ultimo elemento <Peek>: 6
Presione una tecla para continuar . . .
```

Este par de colecciones deben ser usadas cuando nos interesa tener control sobre el orden en que los elementos son obtenidos de las mismas. La pila se debe usar cuando queremos obtener los elementos en el orden inverso al cual fueron ingresados. Mientras que la cola debe ser utilizada cuando queremos obtener los elementos en el mismo orden que fueron ingresados.

## HashTable

Una tabla hash o Hash Table, es una colección que permite almacenar pares de objetos, donde el primero es conocido como llave (key) y el segundo es conocido como valor (value). De esta manera, para agregar “un” elemento a la colección, siempre se debe pasar como argumento estos dos objetos.

```
Hashtable tabla = new Hashtable();  
tabla.Add(2, 3);  
tabla.Add(4, 9);
```

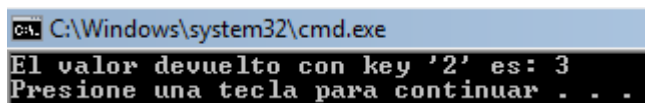
Una de las ventajas que presenta ésta colección frente a las colecciones mencionadas anteriormente ([Parte I: Vectores](#), [Parte II: ArrayList](#), [Parte III: Stack y Queue](#)), es el poco tiempo que tarda en realizar la búsqueda de un elemento determinado, debido a que utiliza el hash de las llaves (keys) para ordenar los elementos, permitiendo así, tener una ubicación única para cada uno.

Un Hash consiste en pocas palabras, en un código único generado por una función Hash, la cual genera una salida única para entradas diferentes y además, produce la misma salida para entradas iguales. Por estas características de una función Hash, no pueden haber llaves iguales en la tabla Hash ya que esto produciría el mismo Hash y no sería posible diferenciar unívocamente los elementos de la colección.

El comportamiento de la HashTable se puede comparar a una tabla de una base de datos relacional, donde cada tabla generalmente (en este caso obligatoriamente) tiene una llave primaria por medio de la cual se diferencia cada registro de los demás y que además su valor debe ser único en toda la tabla. De la misma manera se comporta la HashTable ya que los elementos son ordenados según su Hash.

Cuando sea necesario obtener uno de los elementos contenidos en la HashTable, se debe utilizar la llave o key con la que fue almacenada, como se muestra a continuación:

```
Hashtable tabla = new Hashtable();  
tabla.Add(2, 3);  
tabla.Add(4, 9);  
object valor = tabla[2];  
Console.WriteLine("El valor devuelto con key '2' es: "+valor);
```



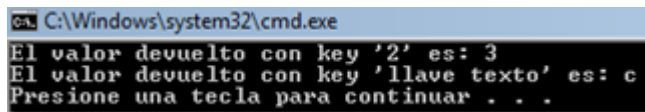
C:\Windows\system32\cmd.exe  
El valor devuelto con key '2' es: 3  
Presione una tecla para continuar . . .

Siempre que se referencia la HashTable utilizando una Key, será retornado el valor correspondiente a dicha clave en un objeto de tipo Object, por lo que es importante

convertir el dato obtenido al tipo de dato adecuado para evitar generar excepciones. Lo anterior, permite tener en la colección pares de llaves y valor que pertenezcan a tipos de datos diferentes, es decir, nada nos obliga a que todos los elementos del HashTable sean del mismo tipo. Incluso las llaves (keys) y los valores (values), pueden ser de tipos de datos diferentes entre si. Sin embargo, no es muy recomendable utilizar la HashTable de esta manera, ya que eso implicaría conocer exactamente que elementos están incluidos en la HashTable e implicaría tener mucho mas cuidado a la hora de extraer los datos, lo cual impediría que la aplicación sea flexible y escalable.

```
Hashtable tabla = new Hashtable();
tabla.Add(2, 3);
tabla.Add(4, null);
tabla.Add(2f, "Valor texto");
tabla.Add("llave texto", 'c');

object valor = tabla[2];
Console.WriteLine("El valor devuelto con key '2' es: "+valor);
valor = tabla["llave texto"];
Console.WriteLine("El valor devuelto con key 'llave texto' es: " + valor);
```



```
C:\Windows\system32\cmd.exe
El valor devuelto con key '2' es: 3
El valor devuelto con key 'llave texto' es: c
Presione una tecla para continuar . . .
```

En el ejemplo anterior se ve como se puede ingresar llaves y valores de diferentes tipos de datos. Además se puede ver como es posible ingresar un valor de tipo null, lo cual no es posible hacerlo en una de las llaves ya que se generaría la excepción `ArgumentNullException`.

Utilizando la sintaxis “[key]” (por ejemplo: `tabla["llave texto"]`) sobre una tabla Hash, es posible actualizar el valor contenido o en caso de que la llave utilizada no exista en la tabla, se inserta este nuevo elemento.

```
Hashtable tabla = new Hashtable();
tabla.Add(2, 3);
tabla.Add(4, null);
tabla["nueva"] = 8;
```

La clase `HashTable` pertenece al espacio de nombres `System.Collections` e implementa las colecciones `IEnumerable`, `ICollection`, entre otras. Debido a esto, es posible utilizar la instrucción `foreach` para leer datos de una `HashTable`. Sin embargo, en cada iteración de esta instrucción, se debe operar con un objeto de tipo `DictionaryEntry` que contendrá un par llave/valor.

```
foreach (DictionaryEntry de in tabla)
{
    Console.WriteLine("Llave: "+de.Key+ " - Valor: "+de.Value);
}
```

Algunos de los métodos mas utilizados de la clase `HashTable` son:

**Add:** permite agregar un nuevo par llave/valor a la colección.

**Remove:** permite quitar un par llave/valor de la colección

**ContainsKey:** permite saber si la colección contiene un par cuya clave sea la que se le pasa como parámetro.

**ContainsValue:** permite saber si la colección contiene un par cuyo valor sea el que se le pasa como parámetro.

## Colecciones genéricas

A diferencia de las colecciones de datos tratadas en los post anteriores, las colecciones genéricas se encuentran en el namespace `System.Collections.Generic` ya que son colecciones con la misma funcionalidad que las colecciones no genéricas (las normales, las que están en el namespace `System.Collections`), con la diferencia que estas están orientadas a trabajar con un tipo de dato específico.

Con un tipo de dato específico?. Si así es, porque a diferencia de las colecciones no genéricas en las que se pueden agregar cualquier tipo de elementos y todos son convertidos a `System.Object`, en este tipo de colecciones solo es posible almacenar elementos de un tipo de dato específico.

Cuando comencé a leer sobre esta nueva característica ofrecida por el Framework 2.0, me preguntaba si el nombre verdaderamente correspondía a la definición de estas colecciones, ya que no entendía el porque llamarlas genéricas si solo era posible almacenar datos del mismo tipo. Por el contrario, parecían ser más genéricas las otras colecciones (las del namespace `System.Collections`) ya que me permitían almacenar cualquier tipo de dato.

Pero finalmente y después de leer varios artículos y códigos de ejemplo, comprendí el porque del nombre `generics` y espero poder transmitirlo en este post.

Definitivamente, una colección genérica solo puede almacenar datos de un tipo, pero dicho tipo de dato es definido por nosotros mismos al momento de declararla e instanciarla. Es decir, no hay una clase para diferenciar las colecciones que operan con enteros de las que operan con cadenas de texto ni de las que operan objetos de tipo `Empleado` (por ejemplo). Absolutamente todas están definidas en la misma clase, lo único que las diferencia es la manera en que se instancia y se declara cada una de ellas. Ese es el secreto del ser genérica. Para entender mejor el concepto de la colección genérica, veamos el siguiente ejemplo:

```
List<int> listaEnteros = new List<int>();  
  
List<string> listacadenas = new List<string>();  
  
List<TablaDatos> listaTablas = new List<TablaDatos>();
```

En el ejemplo se observa la implementación de la colección genérica `List`, que es la versión genérica del `ArrayList` tratado en un post anterior. Se puede deducir que la primera colección permitirá almacenar únicamente datos de tipo `int`, que la segunda almacenara solo cadenas de texto y que la última almacenara objetos de tipo

TablaDatos, el cual es un tipo definido en una clase aparte. En la definición de las tres colecciones se utilizó la clase List, es decir, que hay una clase “genérica” capaz de trabajar con el tipo de dato que indiquemos al momento de inicializarla. (Realmente no es cualquier tipo de dato, sino los tipos de datos soportados por la clase).

La diferencia para inicializar una clase genérica de una clase no genérica, es que en la primera se debe utilizar un parámetro adicional después del nombre de la clase y entre los caracteres < y >. Este parámetro debe ser un tipo de datos, que indica el tipo de datos con el que la colección deberá operar.

Así como esta la colección List<T>, que es la versión genérica del ArrayList, también existen las colecciones genéricas Stack<T>, Queue<T> y Dictionary<T>, entre otras que son las versiones genéricas de las clases Stack, Queue y Hashtable respectivamente.

Un ejemplo de la clase genérica de Stack sería:

```
Stack<int> pilaTexto = new Stack<int>();  
pilaTexto.Push(3);  
pilaTexto.Push(15);  
int temp = pilaTexto.Pop() * 2;
```

Se puede ver como se define una pila en la que únicamente se pueden almacenar datos de tipo entero, lo cual me permite obtener datos de la colección y tratarlos directamente como enteros sin necesidad de hacer algún tipo de cast, lo cual no sería posible con las colecciones no genéricas ya que el elemento es almacenado como un System.Object.

El hecho de que las colecciones genéricas operen con un tipo de dato definido en el momento de su declaración, las hace mucho más eficientes, ya que evita tener que realizar boxing, unboxing y casting de objetos. Adicionalmente los desarrolladores, obtienen un mayor grado de control sobre la información almacenada en la colección ya que en caso de intentar ingresar un objeto de un tipo de dato diferente al establecido en la inicialización de la colección, se arroja una excepción.

Además de esta ventaja que nos brindan las colecciones genéricas, tenemos la posibilidad de crear nuestras propias clases y métodos genéricos cuyo comportamiento estará ligado al tipo de dato que se utilice al utilizarlo tal y como sucede con las colecciones. Para lograr esto debemos hacer uso de los “type parameters”, los cuales permiten establecer un parámetro cuyo tipo se desconocerá hasta que se indique en la declaración.

Para comprender mejor el funcionamiento de los “type parameters” que nos permiten crear clases genéricas, he desarrollado una aplicación de muestra que consiste en una clase impresora “genérica”, que es capaz de almacenar en una cola, objetos de un tipo específico (del tipo que se define al momento de definir un objeto de esta clase). Adicionalmente expone un método “imprimir” que se encarga de imprimir en pantalla cada uno de los valores contenidos en la cola. A continuación presento los fragmentos principales del código.



```

/// <summary>
/// Representa una impresora clasica, con posibilidad de encolar elementos
/// para luego imprimirlos.
/// Felipe Zapata Ramirez. http://developmentmania.wordpress.com
/// </summary>
/// <typeparam name="T"></typeparam>
class Impresora<T>
{
    //Atributos
    private string tipo;
    private Queue<T> cola;

    //Constructor
    public Impresora()
    {
        this.tipo = this.GetType().ToString();
        cola = new Queue<T>();
    }
}

```

En esta primera imagen, se observa como se ha definido una clase Impresora genérica. Note que en este caso se utiliza la letra T para representar el “type parameter”, sin embargo es posible utilizar cualquier letra o palabra. Mediante este parámetro estamos indicando que al momento de definir e instanciar un objeto de tipo Impresora, se debe pasar como parámetro entre < y > el tipo de datos con el que se desea trabajar.

En el constructor de esta clase, se puede ver como la Queue genérica que tiene la clase Impresora como atributo toma su mismo tipo de dato, es decir, si al crear la clase Impresora indicamos el tipo de dato int, la Queue también tomara el tipo de dato int.

```

//Propiedad para la cola
public Queue<T> Cola
{
    get
    {
        return cola;
    }
}

/// <summary>
/// Imprime en pantalla cada uno de los valores contenidos en la cola
/// </summary>
public void imprimir()
{
    Console.WriteLine();
    Console.WriteLine("Imprimiendo cola de tipo: " + tipo);
    Console.WriteLine();

    while (Cola.Count > 0)
    {
        Console.WriteLine(Cola.Dequeue() + " es de tipo : " + tipo);
    }
}

```

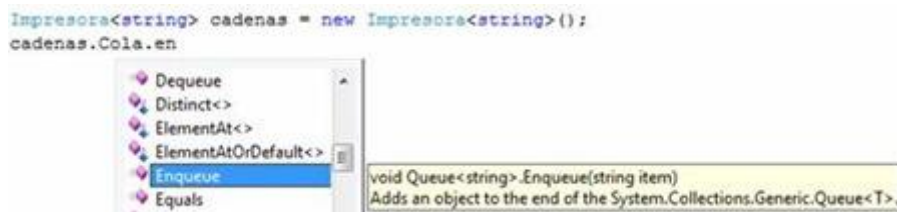
En esta segunda imagen se observa una propiedad que retorna una cola genérica, pero el type parameter para indicar que es una Queue genérica es el mismo utilizado en nuestra clase Impresora, lo cual indica que retornara una Queue del tipo de dato utilizado en la clase.

Finalmente se observa el método imprimir, cuyo objetivo es indicar el tipo de Queue que se esta imprimiendo y además mostrar cada uno de los elementos que contiene.

Como paso final, para comprobar el funcionamiento de la clase Impresora, se crea un objeto de tipo Impresora de enteros, así:

```
Impresora<int> enteros = new Impresora<int>();  
enteros.Cola.Enqueue(2);  
enteros.Cola.Enqueue(3);  
enteros.Cola.Enqueue(4);  
enteros.imprimir();  
  
Impresora<string> cadenas = new Impresora<string>();  
cadenas.Cola.Enqueue("Hola");  
cadenas.Cola.Enqueue("Mundo");  
cadenas.imprimir();
```

Otra de las ventajas que nos brinda la utilización de Generics, es que los “type parameters” se traducen en tiempo de compilación, lo cual implica que es posible darnos cuenta en el momento que desarrollamos nuestra aplicación, el tipo de datos con el que se esta trabajando en cada instancia de la clase Impresora, como se ve a continuación:



Finalmente, se observa el resultado de la ejecución de la aplicación en consola:

```
C:\Windows\system32\cmd.exe  
Imprimiendo cola de tipo: generics.Impresora`1[System.Int32]  
2 es de tipo :generics.Impresora`1[System.Int32]  
3 es de tipo :generics.Impresora`1[System.Int32]  
4 es de tipo :generics.Impresora`1[System.Int32]  
  
Imprimiendo cola de tipo: generics.Impresora`1[System.String]  
Hola es de tipo :generics.Impresora`1[System.String]  
Mundo es de tipo :generics.Impresora`1[System.String]  
Presione una tecla para continuar . . . _
```

En conclusión, debemos utilizar las colecciones genéricas tanto como sea posible en lugar de utilizar las colecciones no genéricas, ya que las primeras incrementan el rendimiento de la aplicación ya que evita un sin numero de casting, boxing y unboxing lo que permite realizar un mejor manejo de la memoria y procesamiento.

Además podemos crear propias clases y/o métodos genéricos para aumentar la flexibilidad y la posibilidad de reutilizar nuestro código.