Tema 4 – Colecciones y Enumeraciones

ARRAYS

Un array es un conjunto de elementos de un cierto tipo, a los cuales es posible acceder por medio de un índice.

```
float[] notas = new float[5];
```

Después de declarar el array, en la definición, hay que indicar la cantidad de elementos que poseerá usando new y especificando el tipo de dato.

Inicializando elementos de arrays

Se pueden inicializar los valores de un array de varios modos:

Usando los subíndices:

```
float[] notas = new float[3];
notas[0] = 5.0f;
notas[1] = 7.5f;
notas[2] = 8.3f;
```

o de manera directa:

```
int[] nums = new int[3] {3, 4, 7} ;

int[] nums2 = {5, 9, 2, 1} ;
```

Array que contiene diferentes tipos de de datos

Para crear un array que pueda contener objetos de diversos tipo, hay que crear un array de objetos de tipo Object. Podremos agregar números, strings, booleanos o cualquier objeto.

```
Object[] elementos = new Object[3];
elementos[0] = "cadena";
elementos[1] = 30;
elementos[2] = false;
```

Recorriendo arrays

Los arrays derivan implícitamente de la clase Array, lo cual brinda métodos y propiedades para su manipulación. Una de ellas es Length, que tiene como función devolver el número total de elementos que posee el array.

De esta manera combinando length con un bucle for se puede recorrer un array:

```
float sumaDeNotas = 0;
for(int i=0; i<notas.Length; i++){
	sumaDeNotas += notas[i];
}
float promedio = sumaDeNotas / notas.Length;
```

Operadores muy usados en bucles

Operador	Ejemplo	Expresión equivalente
+=	x+=y;	x=x+y;
-=	x-=y;	x=x-y;
*=	x*=y;	x=x*y;
/=	x/=y;	x=x/y;
++	x++;	x=x+1;
--	y--;	x=x-1;

La sentencia que mejor se adecua para leer los arrays es foreach en donde se declara una variable que oficiará de elemento en cada iteración.

```
int[] nums2 = {5, 9, 2, 1} ;
foreach(int i in nums2){
      Console.WriteLine(i);
}
```

Métodos y propiedades de la clase Array

Método	Estático	Descripción
BinarySearch	Sí	Búsqueda binaria de un elemento en el array ordenado
Clear	Sí	Limpia un array completo o en parte.
Copy	Sí	Copia la cantidad de elementos especificada de un arrar a otro
CopyTo	No	Copia desde el elemento especificado de un array a otro.
GetEnumerator	No	Devuelve un Enumerator para el array.
GetLength	No	Devuelve tamaño del array.
GetValue	No	Devuelve un elemento en el índice especificado por parámetro.
Reverse	Sí	Invierte el orden de los elementos de un array.
SetValue	No	Fija un elemento del array en el índice especificado por parámetro.
Sort	Sí	Ordena el contenido de un array.

Propiedad	Descripción
IsFixedSize	Indica si el array posee un tamaño fijo
IsReadOnly	Indica si el array es de sólo lectura.
IsSynchronized	Indica si el array puede ser accedido de modo seguro por diversos threads en modo concurrente
Length	Tamaño del array
Rank	Indica la cantidad de dimensiones de un array.

```
public static void Main(){
	//se ordena y se busca un valor en particular
	int[] nums = {5, 9, 3, 1, 30, 15, 11, 6, 5, 2} ;

	//ordeno array
	Array.Sort(nums);

	for(int i=0; i<nums.Length; i++){
		Console.WriteLine("{0} : {1}", i, nums[i]);
	}

	//busqueda de elemento arbitrario
	int elemento = Array.BinarySearch(nums, 30);

	if(elemento >= 0){
		Console.WriteLine("Elemento encontrado en indice: {0}", elemento);
	}else{
		Console.WriteLine("Elemento no encontrado");
	}
}
```

El método BinarySearch devuelve el índice del elemento encontrado, y en caso contrario, un valor negativo.

```
public static void Main(){
	int[] nums = {5, 9, 3, 30, 15} ;
	int[] nums2 = new int[nums.Length];
	//copia de contenido de un array a otro
	nums.CopyTo(nums2, 0);
	//inviertir uno de los arrays
	Array.Reverse(nums2);

	Console.WriteLine("array nums");
	for(int i=0; i<nums.Length; i++){
		Console.WriteLine("{0} : {1}", i, nums[i]);
	}
	Console.WriteLine("array nums2");
	for(int i=0; i<nums2.Length; i++){
		Console.WriteLine("{0} : {1}", i, nums2[i]);
	}
}
```

El primer parámetro que espera el método CopyTo es el array destino, y el segundo parámetro, el índice a partir del cual comenzar la copia.

Arrays como parámetros de métodos

Cuando se pasa un array como parámetro, no se está pasando una copia de éste, sino una referencia a él, lo que permite a una función modificar directamente su contenido.

```
public static void ModificarArray(int[] arr){
	arr[0] = 50;
}
int[] nums = {5, 9, 3, 30, 15} ;
MainClass.ModificarArray(nums);
```

Si se desea pasar en forma de parámetro un elemento del array:

```
public static void ModificarElemento(ref int num){
	num = 33;
}

int[] nums = {5, 9, 3, 30, 15} ;
MainClass.ModificarElemento(ref nums[0]);
```

El hecho de utilizar ref como modificador del parámetro es para especificar que dicho parámetro debe ser pasado como referencia.

Valores predeterminados en un array

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

En el ejemplo anterior, el array comenzará el relleno con ceros, se inicializa automáticamente con este valor.

Pero si el array es de un tipo referenciado (array de objetos), entonces los elementos son inicializados en null, y para poder hacer uso de cada uno de ellos se deberán inicializar individualmente por medio del operador new.

```
class MyClass{
	public MyClass(){
		Console.WriteLine("Constructor");
	}
}
class MainClass{
	public static void Main(){
		//array de objetos
		MyClass[] arrayObjects = new MyClass[5];
		for(int i=0; i<arrayObjects.Length; i++){
			arrayObjects[i] = new MyClass();
		}
	}
}
```

Declarar el array de objetos no produce la invocación de los constructores, el array de objetos tendrá cinco elementos con valor null (objetos sin inicializar). Para poder utilizar los elementos deberemos inicializarlos de forma individual.

El modificador params

Es posible especificar un modificador al parámetro de una función llamado **params**, que permitirá pasar arrays o un conjunto de elementos que el compilador traducirá a arrays de manera dinámica.

```
class MainClass{
	static void Listar(params int[] nums){
		for(int i=0; i<nums.Length; i++){
			Console.WriteLine(nums[i]);
		}
	}
```

```
	public static void Main(){
		int[] nums = {3, 1, 8, 5, 9} ;
		//array convencional
		MainClass.Listar(nums);
		//array creado dinamicamente
		MainClass.Listar(1, 2, 3, 99);
	}
}
```

ARRAYS MULTIDIMENSIONALES

Se pueden definir arrays de dos o varias dimensiones. Por ejemplo un array de dos dimensiones.

```
int[,] tabla = new int[3,3];
```

En lugar de haber un solo número en la indicación del tamaño del array, figuran dos, correspondientes a la cantidad de elementos por cada dimensión. Un array de dos dimensiones se puede imaginar como una tabla, siendo la primera dimensión la cantidad de filas, y la segunda dimensión, la cantidad de columnas.

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

Para poder acceder a los elementos de un array de dos dimensiones, se deberá hacer uso de la siguiente sintaxis.

```
tabla[0,0] = 10;
```

Toda implementación en arrays multidimensionales puede realizarse en arrays de una dimensión. Un array de 3x3 sería el equivalente a un array de 9 elementos de una sola dimensión.

Ejemplo, rellenar con valores un array de dos dimensiones usando un bucle y después mostrar esos valores.

```
public static void Main(){
	int filas = 3;
	int columnas = 3;
	int[,] tabla = new int[filas, columnas];

	//rellenar de valores el array
	for(int i=0; i<filas*columnas; i++){
		tabla[i/columnas, i%columnas] = i;
	}

	//mostrar valores array
	for(int fila=0; fila<filas; fila++){
		for(int col=0; col<columnas; col++){
			Console.WriteLine("({0},{1}) = {2}",fila, col, tabla[fila, col]);
		}
	}
}
```

En el bucle que rellena de valores el array de dos dimensiones hay:

```
tabla[i/columnas, i%columnas] = i;
```

Esta sentencia funciona de la siguiente manera en cada ciclo del bucle:
% es el módulo de la división (resto de la división)

- para i = 0 y columnas = 3
  - i/columnas = 0
  - i%columnas = 0
- para i = 1 y columnas = 3
  - i/columnas = 0
  - i%columnas = 1
- para i = 2 y columnas = 3
  - i/columnas = 0
  - i%columnas = 2
- para i = 3 y columnas = 3
  - i/columnas = 1
  - i%columnas = 0

Y así sucesivamente...

En el caso de representar una tabla con un array de una dimensión:

```
public static void Main(){
	int filas = 3;
	int columnas = 3;
	int[] tabla = new int[filas * columnas];

	//rellenar de valores el array
	for(int i=0; i<filas*columnas; i++){
		tabla[i] = i;
	}

	//mostrar valores array
	for(int fila=0; fila<filas; fila++){
		for(int col=0; col<columnas; col++){
		Console.WriteLine("({0},{1}) = {2}",fila, col, tabla[fila*columnas+col]);
		}
	}
}
```

La parte `tabla[fila*columnas+col]` funcionaria de la siguiente manera para acceder a los índices del array de una dimensión usando el numero de filas y columnas:

```
0*3+0 = 0
0*3+1 = 1
0*3+2 = 2
1*3+0 = 3
1*3+1 = 4
1*3+2 = 5
2*3+0 = 6
2*3+1 = 7
2*3+2 = 8
```

Para inicializar un array de dos dimensiones en el momento de definirlo:

```
int[,] tabla = {{0,1,2} ,{3,4,5} ,{6,7,8}} ;
```

ARRAYS DE ARRAYS

También llamados jagged arrays, se trata de un array en el que sus elementos son otros arrays.

```
int[][] arrayDeArrays;
```

Para definir el array de nivel más alto, el que contendrá al resto de arrays:

```
arrayDeArrays = new int[2][];
```

Y para definir los arrays que se encuentran dentro:

```
arrayDeArrays[0] = new int[2];
arrayDeArrays[1] = new int[3];
```

Para acceder a los elementos:

```
arrayDeArrays[0][0] = 1;
```

También es posible inicializar el array en el momento de definirlo:

```
float[] alumno1 = {5.7f, 7.8f, 6.5f} ;
float[] alumno2 = {9.7f, 5.8f, 8.5f} ;
float[][] notas = {alumno1, alumno2};
```

Para recorrerlo se puede usar un bucle foreach anidado:

```
//mostrar valores array notas
int contador = 1;
foreach(float[] alumno in notas){
	Console.WriteLine("Alumno{0}", contador);
	foreach(float nota in alumno){
		Console.WriteLine("nota {0}", nota);
	}
	contador++;
}
```

INDEXADORES también llamados INDIZADORES

Los indexadores permiten acceder a una colección dentro de un objeto que se haya creado como si éste fuese un array. Un indexador se asemeja a una propiedad, pero no posee nombre, sino que es invocado cuando se utiliza el operador de indexación [] junto con el identificador del objeto.

Un indexador se define como una propiedad, excepto que:

- El nombre siempre es la palabra clave this.
- A this le sigue una lista de parámetros encerrada entre corchetes. Todos los parámetros deben pasarse por valor.

Ejemplo de clase Cadena, para uso y manipulación de strings en donde para acceder a la variable privada texto se ha creado una propiedad de acceso.

```
class Cadena{
	//declaracion de array de caracteres
	private char[] texto;

	//propiedad de acceso a texto
	public string Texto{
		get{
			string str = "";
			//se extrae cada caracter del array
			//y se va concatenando en un string
			foreach(int c in this.texto)
				str += (char) c;
			return str;
		}
		set{
			//se inicializa el array de caracteres
			//con el tamaño de la cadena pasada
			this.texto = new char[value.Length];
			//se introduce cada caracter de la
			//cadena en el array
			for(int i=0; i<value.Length; i++)
				this.texto[i] = value[i];
		}
	}
}
```

Ahora se desea permitir que a través del objeto de tipo Cadena se pueda acceder a cada letra del string(elementos del array texto) como si el objeto fuese un array. Para ello se añade dentro de la clase Cadena un indexador.

```
//indexador de acceso a array texto
public char this[int index]{
	get{
		if(index < this.texto.Length)
			return this.texto[index];
		else
			return '\0';
	}
	set{
		if(index < this.texto.Length)
			this.texto[index] = value;
	}
}
```

A la hora de usar el indexador y la propiedad en el método Main se procede:

```
class MainClass{
	public static void Main(){
		Cadena cad = new Cadena();

		//uso de set la propiedad Texto
		cad.Texto = "el lenguage C#";

		//uso de get de la propiedad Texto
		Console.WriteLine(cad.Texto);
```

```
            //uso de get del indexador
            Console.WriteLine(cad[12]);

            //uso combinado de propiedad e indexador
            for(int i=0; i<cad.Texto.Length; i++)
                  Console.Write(cad[i]);

            //uso de set del indexador
            cad[0] = 'E';
            cad[3] = 'L';
            Console.WriteLine();

            //uso combinado de propiedad e indexador
            for(int i=0; i<cad.Texto.Length; i++)
                  Console.Write(cad[i]);
      }
}
```

Se hace uso del indexador definido en la clase con la finalidad de leer los caracteres del array que se encuentra encapsulado en su interior, así como para modificar su contenido.

Indexadores para acceder a arrays de dos dimensiones

```
class Coordenada{
      //declaracion de array 2d
      private int[,] posiciones;

      //constructor
      public Coordenada(int id1, int id2){
            this.posiciones = new int[id1, id2];
      }

      //indexador de acceso a array 2d
      public int this[int idx1, int idx2]{
            get{
                  return this.posiciones[idx1, idx2];
            }
            set{
                  this.posiciones[idx1, idx2] = value;
            }
      }
}

class MainClass{
      public static void Main(){
            Coordenada pos1 = new Coordenada(5,7);
            pos1[3,4] = 10;
            Console.WriteLine(pos1[3,4]);
      }
}
```

Que un indexador permita recibir más de un tipo de dato no significa que se deba utilizar solamente para el acceso a arrays multidimensionales. Tampoco es necesario que todos los elementos que reciba el indexador sean del mismo tipo.

INTERFACES PARA COLECCIONES

Se puede usar foreach en clases si estas implementan la interface IEnumerable. La interface IEnumerable contiene métodos para extraer elementos de los objetos. **Si la clase contiene una lista de elementos y se quiere que mediante foreach iterar cada uno de los elementos, hay que implementar la interface IEnumerable.**

Una colección se utiliza para trabajar con listas o conjuntos ordenados de objetos. Esta funcionalidad proviene de la implementación de una serie de interfaces del **namespace System.Collections**.

Implementar la funcionalidad de estas interfaces permite crear una colección personalizada y acceder a los elementos de la colección utilizando corchetes, de manera idéntica a como se accede a los elementos de un array.

IEnumerable y IEnumerator

IEnumerable: Proporciona la funcionalidad para "recorrer" una objeto por medio de una sentencia foreach. Al implementar IEnumerable hay que implementar el método:

```
IEnumerator GetEnumerator();
```

Además de una clase que implemente IEnumerable, se necesita otra clase auxiliar que implemente IEnumerator. La responsabilidad de la clase que implemente IEnumerator será la de proporcionar los métodos necesarios para recorrer una colección: Current, MoveNext() y Reset()

```
object Current{get;}
```

propiedad que será utilizada cuando se desee acceder al elemento actual:

También se declaran los siguientes métodos:

```
bool MoveNext();
void Reset();
```

MoveNext traslada el cursor al siguiente elemento de la colección.
Reset devuelve el cursor al inicio de la colección.

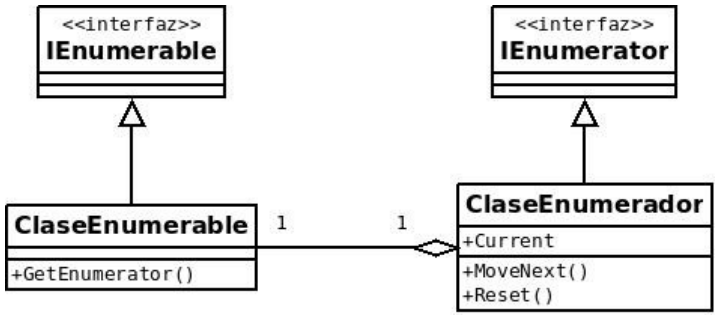
Un enumerador es un objeto que apunta a un elemento de una colección. Es un cursor que permite recorrer una lista pasando de un elemento a otro mediante un método. Los enumeradores permiten recorrer una colección de un mismo modo, más allá de cómo estén implementados.

Interfaz	Métodos/propiedades
IEnumerable	`IEnumerator GetEnumerator();`

IEnumerator	`object Current{get;}`
	`bool MoveNext();`
	`void Reset();`

Estos métodos se implementan para admitir el uso de foreach para recorrer en iteración la colección.

Ejemplo:

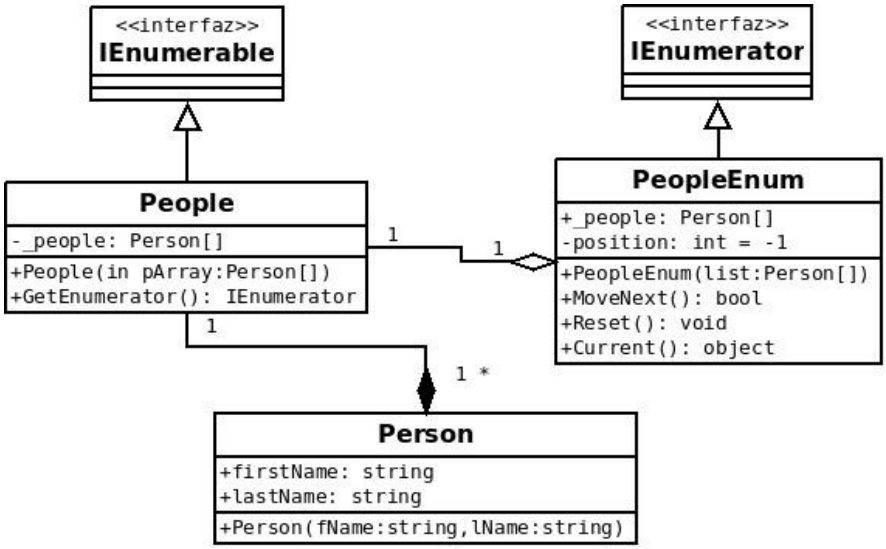



```
using System.Collections;
using System;

namespace ejemplo_14
{
	class ClaseEnumerable : IEnumerable{
		//lista de enteros que se recorrrera
		private int[] lista = {0, 1, 2, 3, 4, 5} ;

		//metodo implementado del interfaz IEnumerable
		//retorna un enumerador
		public IEnumerator GetEnumerator(){
			return new ClaseEnumerador(this);
		}

		//indexador para acceso a
		//elementos del array
		public int this[int idx]{
			get{
				return lista[idx];
			}
		}

		//propiedad
		public int Length{
			get{
				return lista.Length;
			}
		}
	}

	class ClaseEnumerador : IEnumerator{
		//referencia al objeto a recorrer
		private ClaseEnumerable objEnum = null;
		//se coloca antes del primer elemento
		private int elementoActual = -1;

		//constructor
		public ClaseEnumerador(ClaseEnumerable obj){
			objEnum = obj;
		}
```

```
            //propiedad retorna elemento actual
            public object Current{
                  get{
                        return objEnum[elementoActual];
                  }
            }

            public bool MoveNext(){
                  if(elementoActual < objEnum.Length-1){
                        elementoActual++;
                        return true;
                  }else
                        return false;
            }

            public void Reset(){
                  elementoActual = -1;
            }
      }

      class MainClass{
            public static void Main(){
                  ClaseEnumerable objEnum = new ClaseEnumerable();
                  foreach(int i in objEnum)
                        Console.WriteLine(i);
            }
      }
}

private int elementoActual = -1;
```

Inicialmente, el enumerador se coloca antes del primer elemento de la colección. El método Reset también devuelve el enumerador a esta posición. En esta posición, una llamada a la propiedad Current produce una excepción. Por lo tanto, se debe llamar al método MoveNext para desplazar el enumerador hasta el primer elemento de la colección antes de leer el valor de Current.

Otro ejemplo, esta vez la clase enumerable guardará un array de objetos de otra clase.



```
using System.Collections;
using System;

namespace ejemplo_15
{
	class Person{
		public string firstName;
		public string lastName;

		public Person(string fName, string lName){
			this.firstName = fName;
			this.lastName = lName;
		}
	}

	class People : IEnumerable{
		//array de objetos de clase Person
		private Person[] _people;

		//constructor
		public People(Person[] pArray){
			//inicializa array
			_people = new Person[pArray.Length];

			//rellena array con objetos
			for (int i = 0; i < pArray.Length; i++){
			_people[i] = pArray[i];
			}
		}

		public IEnumerator GetEnumerator(){
			return new PeopleEnum(_people);
		}
	}

	class PeopleEnum : IEnumerator{
		public Person[] _people;
		int position = -1;

		public PeopleEnum(Person[] list){
			_people = list;
		}

		public bool MoveNext(){
			position++;
			return (position < _people.Length);
		}

		public void Reset(){
			position = -1;
		}

		public object Current{
			get{
				try{
				return _people[position];
				}
				catch (IndexOutOfRangeException){
					throw new InvalidOperationException();
				}
			}
		}
	}
```

```
	class App{
		static void Main(){
			//array de objetos de la clase Person
			Person[] peopleArray = new Person[3]
			{
			new Person("John", "Smith"),
			new Person("Jim", "Johnson"),
			new Person("Sue", "Rabon"),
			} ;

			People peopleList = new People(peopleArray);
			foreach (Person p in peopleList)
				Console.WriteLine(p.firstName + " " + p.lastName);

		}
	}
}
```

Se puede recorrer una colección de datos a través de una referencia a un objeto que implemente IEnumerable por ejemplo un string o array de strings ya que son objetos que implementan IEnumerable:

```
class App{
	static void Main(){
		string s = "abc123";
		Console.WriteLine("el string s es enumerable? "+
					(s is IEnumerable)+"\n");

		foreach (char c in s) Console.WriteLine(c);

		Console.WriteLine("\n---\n");

		IEnumerator ie = s.GetEnumerator();

		while (ie.MoveNext()) {
			char c = (char) ie.Current;
			Console.WriteLine(c);
		}
	}
}
```

Ejemplo de clase que implementa las dos interfaces

```
class Alumnos : IEnumerable, IEnumerator{
	int position = -1;

	public IEnumerator GetEnumerator(){
		return this;
	}

	public object Current{
		get{
			switch(position){
				case 0:
					return "Jose";
				case 1:
					return "Paco";
				case 2:
					return "Maria";
				case 3:
					return "Ana";
```

```
                        default:
                                return "Error";
                }
        }
}

public bool MoveNext(){
        position++;
        if(position == 4)
                return false;
        return true;
}

public void Reset(){
        position = -1;
}

public static void Main(){
        Alumnos MyClase = new Alumnos();
        foreach(string Alumno in MyClase)
                Console.WriteLine(Alumno);
}
}
```

ICollection

Interface que hereda de IEnumerable. Proporciona la capacidad para obtener el número de elementos de la colección y de copiar elementos a un simple array.

Interface	Métodos

ICollection	`int Count {get;}`
	`bool IsSynchronized {get;}`
	`object SyncRoot {get;}`
	`void CopyTo(Array array, int index)`

`int Count {get;}` : Devuelve la cantidad total de elementos que posee la colección.

`bool IsSynchronized {get;}` : Indica si el acceso al objeto es seguro por medio de diferentes threads concurrentes.

`object SyncRoot {get;}` : Esta propiedad devuelve un objeto que permita sincronizar el acceso a la colección.

`void CopyTo(Array array, int index)` : Copia de elementos de la colección al array recibido como parámetro desde el elemento indicado por index.

IComparable

Si una clase implementa la interface Icomparable podrá ser ordenada por medio del método Sort, ya que es posible determinar si un elemento es menor, igual o mayor a otro.

La interface declara sólo el siguiente método:

```
int CompareTo(object obj)
```

Este método recibe un objeto como parámetro. Se espera que el método devuelva -1 si a instancia es menor al objeto, 0 si los dos objetos son iguales y 1 si el objeto es mayor a la instancia.

```
using System;

namespace ejemplo_18
{
	public class Temperature : IComparable {

		// Valor
		protected int t_value;

		//propiedad
		public int Value {
			get {
			return t_value;
			}
			set {
			t_value = value;
			}
		}

		/// <summary>
		/// IComparable.CompareTo implementation.
		/// </summary>
		public int CompareTo(object obj) {
			if(obj is Temperature) {
				Temperature temp = (Temperature) obj;
				return t_value.CompareTo(temp.t_value);
			}
			throw new ArgumentException("object is not a Temperature");
		}

		public static void Main(){
			Temperature t1 = new Temperature();
			t1.Value = 30;
			Temperature t2 = new Temperature();
			t2.Value = 20;

			if(t2.CompareTo(t1) < 0)
				Console.WriteLine("t2 es menor que t1");
			else if(t2.CompareTo(t1) == 0)
				Console.WriteLine("t2 es igual a t1");
			else
				Console.WriteLine("t2 es mayor que t1");
		}
	}
}
```

El objeto pasado como argumento de CompareTo debe ser del mismo tipo que la clase que se compara. También habrá que realizar un casting convertir el objeto al tipo de la clase que se implementa.

```
Temperature temp = (Temperature) obj;
```

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. Si una clase implementa la interface IList, entonces sus elementos podrán ser accedidos individualmente por medio de un índice.

Interface	Métodos/Propiedades

IList	`bool IsFixedSize{get;}`
	`bool IsReadOnly{get;}`
	`object this[int index]{get; set;}`
	`int Add(object value){}`
	`void Clear(){}`
	`bool Contains(object value){}`
	`int IndexOf(object value){}`
	`void Insert(int index, object value){}`
	`void Remove(object value){}`
	`void RemoveAt(int index){}`

`bool IsFixedSize{get;}`: esta propiedad indica si la lista es de un tamaño fijo o si puede crecer dinámicamente.
`bool IsReadOnly{get;}`: indica si la lista es de sólo lectura
`object this[int index]{get; set;}`: este indexador brinda la posibilidad de leer y modificar los elementos de la lista.

`int Add(object value){}`: agrega elementos a la lista
`void Clear(){}`: limpia la lista
`bool Contains(object value){}`: indica si un elemento se encuentra contenido en la lista o no.
`int IndexOf(object value){}`: devuelve el índice al elemento pasado como parámetro, -1 si éste no se encuentra dentro de ella.
`void Insert(int index, object value){}`: este método se encarga de insertar un elemento en una posición determinada.
`void Remove(object value){}`: elimina un elemento de la lista
`void RemoveAt(int index){}`: elimina un elemento de la lista especificando su posición dentro de ésta.

En el ejemplo siguiente, se muestra la implementación de la interfaz IList para crear una lista simple, lista de tamaño fijo

```
using System.Collections;
using System;

namespace ejemplo_19
{
	class Program{
		public static void Main(){
		SimpleList test = new SimpleList();

		// Populate the List
		Console.WriteLine("Populate the List");
		test.Add("one");
		test.Add("two");
		test.Add("three");
```

```
            test.Add("four");
            test.Add("five");
            test.Add("six");
            test.Add("seven");
            test.Add("eight");
            test.PrintContents();
            Console.WriteLine();

            // Remove elements from the list
            Console.WriteLine("Remove elements from the list");
            test.Remove("six");
            test.Remove("eight");
            test.PrintContents();
            Console.WriteLine();

            // Add an element to the end of the list
            Console.WriteLine("Add an element to the end of the list");
            test.Add("nine");
            test.PrintContents();
            Console.WriteLine();

            // Insert an element into the middle of the list
            Console.WriteLine("Insert an element into the middle of the list");
            test.Insert(4, "number");
            test.PrintContents();
            Console.WriteLine();

            // Check for specific elements in the list
            Console.WriteLine("Check for specific elements in the list");
            Console.WriteLine("List contains \"three\": {0}", test.Contains("three"));
            Console.WriteLine("List contains \"ten\": {0}", test.Contains("ten"));
      }
      }  // class Program

      class SimpleList : IList{
            //lista de tamaño fijo
            private object[] _contents = new object[8];
            private int _count;

            public SimpleList(){
                  _count = 0;
            }

            // IList Members

            //returns the position into which the new element was inserted,
            //or -1 to indicate that the item was not inserted into the collection
            public int Add(object value){
                  if (_count < _contents.Length){
                        _contents[_count] = value;
                        _count++;
                        return (_count - 1);
                  }else{
                        return -1;
                  }
            }

            public void Clear(){
                  _count = 0;
            }
```

```
public bool Contains(object value){
	bool inList = false;
	for (int i = 0; i < Count; i++){
		if (_contents[i] == value){
			inList = true;
			break;
		}
	}
	return inList;
}

// returns the index of value if found in the list; otherwise, -1
public int IndexOf(object value){
	int itemIndex = -1;

	for (int i = 0; i < Count; i++){
		if (_contents[i] == value){
			itemIndex = i;
			break;
		}
	}
	return itemIndex;
}

public void Insert(int index, object value){
	if ((_count + 1 <= _contents.Length)
		&& (index < Count) && (index >= 0)){
		_count++;//se incrementa el contador de elementos
		//se reconstruye la lista desde el final hasta
		//la posicion donde se va agregar el nuevo elemento
		//los elementos se van moviendo una posicion hacia el
		//final de la lista
		for (int i = Count - 1; i > index; i--){
			_contents[i] = _contents[i - 1];
		}
		_contents[index] = value;
	}
}

public bool IsFixedSize{
	get{
		return true;
	}
}

public bool IsReadOnly{
	get{
		return false;
	}
}

public void Remove(object value){
	RemoveAt(IndexOf(value));
}
```

```
public void RemoveAt(int index){
	if ((index >= 0) && (index < Count)){
		//se reconstruye la lista desde la posicion de index hasta
		//la penultima posicion
		//los elementos se van moviendo una posicion hacia
		//adelante de la lista
		for (int i = index; i < Count - 1; i++){
			_contents[i] = _contents[i + 1];
		}
		_count--;//se decrementa contador de elementos
	}
}

//indexador
public object this[int index]{
	get{
		return _contents[index];
	}
	set{
		_contents[index] = value;
	}
}

// ICollection Members
//copia elementos de la lista a array
public void CopyTo(Array array, int index){
	int j = index;
	for (int i = 0; i < Count; i++){
		array.SetValue(_contents[i], j);
		j++;
	}
}

//propiedad
public int Count{
	get{
		return _count;
	}
}

public bool IsSynchronized{
	get{
		return false;
	}
}

// Obtiene un objeto que se puede utilizar
// para sincronizar el acceso a ICollection
public object SyncRoot{
	get{
		return this;
	}
}

// IEnumerable Members
public IEnumerator GetEnumerator(){
	throw new Exception("The method or operation is not implemented.");
}
```

```
		//imprime informacion de la lista
		public void PrintContents(){
			Console.WriteLine("List has a capacity of {0} and currently has {1}
elements.", _contents.Length, _count);
			Console.Write("List contents:");
			for (int i = 0; i < Count; i++){
				Console.Write(" {0}", _contents[i]);
			}
			Console.WriteLine();
		}
	}

}
```

CLASES PARA COLECCIONES

Son clases que ya implementan diversos interfaces para trabajar con colecciones.

Clase genérica Collection

La clase Collection implementa los interfaces IList, ICollection y IEnumerable, se puede utilizar de forma inmediata creando una instancia; todo lo que hay que hacer es especificar el tipo de objeto que va a contener la colección.

La clase Collection proporciona métodos que se pueden utilizar para agregar y quitar elementos, borrar la colección o establecer el valor de un elemento existente. Se puede obtener acceso a los elementos de esta colección utilizando un índice de tipo entero.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace ejemplo_20
{
	class Program{
		public static void Main(){
			//especificar el tipo de objeto que va a contener la coleccion
			Collection<string> dinosaurs = new Collection<string>();

			//añadir elementos a la coleccion
			dinosaurs.Add("Psitticosaurus");
			dinosaurs.Add("Caudipteryx");
			dinosaurs.Add("Compsognathus");
			dinosaurs.Add("Muttaburrasaurus");

			//Count obtiene numero de elementos de la coleccion
			Console.WriteLine("{0} dinosaurs:", dinosaurs.Count);
			Display(dinosaurs);

			//IndexOf localiza el indice de un elemento
			Console.WriteLine("\nIndexOf(\"Muttaburrasaurus\"): {0}",
					dinosaurs.IndexOf("Muttaburrasaurus"));

			//Contains determina si un elemento se encuentra en la coleccion
			Console.WriteLine("\nContains(\"Caudipteryx\"): {0}",
					dinosaurs.Contains("Caudipteryx"));
```

```
                        //se inserta un elemento mediante el metodo Insert
                        Console.WriteLine("\nInsert(2, \"Nanotyrannus\")");
                        dinosaurs.Insert(2, "Nanotyrannus");
                        Display(dinosaurs);

                        Console.WriteLine("\ndinosaurs[2]: {0}", dinosaurs[2]);

                        //se cambia el valor de un elemento
                        Console.WriteLine("\ndinosaurs[2] = \"Microraptor\"");
                        dinosaurs[2] = "Microraptor";
                        Display(dinosaurs);

                        //se quitan elementos a partir del valor con Remove
                        Console.WriteLine("\nRemove(\"Microraptor\")");
                        dinosaurs.Remove("Microraptor");
                        Display(dinosaurs);

                        //se quitan elementos a partir del indice con RemoveAt
                        Console.WriteLine("\nRemoveAt(0)");
                        dinosaurs.RemoveAt(0);
                        Display(dinosaurs);

                        //se borran todos los elementos con Clear
                        Console.WriteLine("\ndinosaurs.Clear()");
                        dinosaurs.Clear();
                        Console.WriteLine("Count: {0}", dinosaurs.Count);

                }

                private static void Display(Collection<string> cs){
                        Console.WriteLine();
                        foreach(string item in cs ){
                                Console.WriteLine(item);
                        }
                }
        }
}
```

La clase ArrayList

Una de las clases más importantes que proporciona el namespace Systems.Collections se denomina 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. Es muy parecida a la clase genérica Collection.

Métodos/Propiedades	Explicación
Adapter()	Crea un contenedor de ArrayList para una interfaz IList concreta.
Capacity	Obtiene el número de elementos que puede contener el objeto ArrayList.
Count	Obtiene el número de elementos contenido en ArrayList.
Item	Obtiene o establece el elemento que se encuentra en el índice especificado.
Add()	Agrega al final de ArrayList.
AddRange()	Agrega los elementos de ICollection al final de ArrayList.

Clear()	Quita todos los elementos.
Contains()	Determina si un elemento se encuentra.
IndexOf()	Devuelve el índice de un determinado elemento.
insert()	Inserta un elemento en la clase ArrayList en el índice especificado.
InsertRange()	Inserta los elementos de una colección en ArrayList en el índice especificado.
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.

```
using System;
using System.Collections;

namespace ejemplo_21
{
	class Program{
		public static void Main()  {
			// Creates and initializes a new ArrayList.
			ArrayList myAL = new ArrayList();
			myAL.Add("Hello");
			myAL.Add(" World ");
			myAL.Add("!");

			Console.WriteLine( "Count:    {0}", myAL.Count );
			Console.WriteLine( "Capacity: {0}", myAL.Capacity );
			Console.Write( "Values: " );
			PrintValues( myAL );
		}

		public static void PrintValues(IEnumerable myList)  {
			foreach ( Object obj in myList )
				Console.Write( "{0}", obj );
			Console.WriteLine();
		}
	}
}
```

DICCIONARIOS

Un diccionario es una colección que asocia un valor a una clave. Podría ser considerado como un array que utiliza índices no numéricos para acceder a los elementos. Un ejemplo podría ser querer obtener la referencia a un objeto a partir de su índice no numérico o acceder a la cantidad de días que posee un mes a partir de su nombre.

En un diccionario la clave podrá ser cualquier tipo de dato, lo mismo para el valor.

La clase 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 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 un código único generado por una función Hash. No pueden haber keys iguales en la tabla Hash ya que esto produciría el mismo Hash y no seria posible diferenciar inequívocamente 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 obligadamente) 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(valor);
```

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 recomendable 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 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 un 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, "valor texto");
tabla.Add("llave texto", 9);

object valor = tabla[2];
Console.WriteLine(valor);
valor = tabla["llave texto"];
Console.WriteLine(valor);
```

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, "valor texto");
tabla.Add("llave texto", 9);
tabla["nueva"] = 8;
Console.WriteLine(tabla["nueva"]);
```

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 un HashTable. Sin embargo, en cada iteracion de esta instrucción, se debe operar con un objeto de tipo **DictionaryEntry** que contendrá un par llave/valor.

```
foreach(DictionaryEntry par in tabla){
	Console.WriteLine("LLave: {0} - Valor: {1}", par.Key, par.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.

```
Hashtable openWith = new Hashtable();

// Add some elements to the hash table. There are no
// duplicate keys, but some of the values are duplicates.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");

// The Add method throws an exception if the new key is
// already in the hash table.
try{
	openWith.Add("txt", "winword.exe");
}catch{
	Console.WriteLine("An element with Key = \"txt\" already exists.");
}

Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

// Se cambia valor para la key rtf
openWith["rtf"] = "winword.exe";
Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

// Si una key no existe se añade un par Key/Value nuevo
openWith["doc"] = "winword.exe";
```

```
// The default Item property throws an exception if the requested
// key is not in the hash table.
try{
	Console.WriteLine("For key = \"tif\", value = {0}.", openWith["tif"]);
}catch{
	Console.WriteLine("Key = \"tif\" is not found.");
}

// ContainsKey can be used to test keys before inserting
// them.
if (!openWith.ContainsKey("ht")){
	openWith.Add("ht", "hypertrm.exe");
	Console.WriteLine("Value added for key = \"ht\": {0}", openWith["ht"]);
}

// When you use foreach to enumerate hash table elements,
// the elements are retrieved as KeyValuePair objects.
Console.WriteLine();
foreach( DictionaryEntry de in openWith ){
	Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
}

// Se puede usar la propiedad Values para obtener
// solo los valores y almacenarlos en una variable de
// tipo ICollection
ICollection valueColl = openWith.Values;

Console.WriteLine();
foreach( string s in valueColl ){
	Console.WriteLine("Value = {0}", s);
}

// To get the keys alone, use the Keys property.
ICollection keyColl = openWith.Keys;

Console.WriteLine();
foreach( string s in keyColl ){
	Console.WriteLine("Key = {0}", s);
}

// Use the Remove method to remove a key/value pair.
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc")){
	Console.WriteLine("Key \"doc\" is not found.");
}
```

Las Colecciones Stack & 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 parecen 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.

No es posible acceder de manera aleatoria 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**. 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 primer 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.

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.

```
// Creates and initializes a new Stack.
Stack myStack = new Stack();
myStack.Push("Hello");
myStack.Push("World");
myStack.Push("!");

// Creates and initializes a new Queue.
Queue myQ = new Queue();
myQ.Enqueue("Goodbye");
myQ.Enqueue("Jhon");
myQ.Enqueue("!");
```

Como se ve, para agregar elementos a una pila se debe utilizar el método **Push**. Mientras que en la cola se debe utilizar el método **Enqueue** (encolar).

Para obtener un elemento de la colección, hay dos opciones:

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).

```
// Creates and initializes a new Stack.
Stack myStack = new Stack();
myStack.Push("Hello");
myStack.Push("World");
myStack.Push("!");
```

```
Console.WriteLine("Elementos del stack: {0}", myStack.Count);
Console.WriteLine("Ultimo elemento (Peek): {0}", myStack.Peek());
Console.WriteLine("Elementos del stack: {0}", myStack.Count);
Console.WriteLine("Ultimo elemento (Pop): {0}", myStack.Pop());
Console.WriteLine("Elementos del stack: {0}", myStack.Count);
Console.WriteLine("Ultimo elemento (Peek): {0}", myStack.Peek());

Console.WriteLine();

// Creates and initializes a new Queue.
Queue myQ = new Queue();
myQ.Enqueue("Goodbye");
myQ.Enqueue("Jhon");
myQ.Enqueue("!");

Console.WriteLine("Elementos del Queue: {0}", myQ.Count);
Console.WriteLine("Ultimo elemento (Peek): {0}", myQ.Peek());
Console.WriteLine("Elementos del Queue: {0}", myQ.Count);
Console.WriteLine("Ultimo elemento (Dequeue): {0}", myQ.Dequeue());
Console.WriteLine("Elementos del Queue: {0}", myQ.Count);
Console.WriteLine("Ultimo elemento (Peek): {0}", myQ.Peek());
```

La salida es:

Elementos del stack: 3
Ultimo elemento (Peek): !
Elementos del stack: 3
Ultimo elemento (Pop): !
Elementos del stack: 2
Ultimo elemento (Peek): World

Elementos del Queue: 3
Ultimo elemento (Peek): Goodbye
Elementos del Queue: 3
Ultimo elemento (Dequeue): Goodbye
Elementos del Queue: 2
Ultimo elemento (Peek): Jhon

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 se desee obtener los elementos en el orden inverso al cual fueron ingresados. Mientras que la cola debe ser utilizada cuando se quiera obtener los elementos en el mismo orden que fueron ingresados.

Colecciones Genéricas

Las colecciones genéricas se encuentran en el namespace System.Collections.Generics como la clase genérica Collection 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**.

A diferencia de las colecciones no genéricas en las que se pueden agregar cualquier tipo de elemento y todos son convertidos a System.Object, en este tipo de colecciones solo es posible almacenar elementos de un tipo de dato específico.

Dicho tipo de dato se tiene que definir al momento de declarar e instanciar la colección. 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. Por 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. 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 utiliza la clase List, es decir, que hay una clase genérica capaz de trabajar con el tipo de dato que indiquemos al momento de inicializarla.

Así como la colección List<T>, que es la versión genérica del ArrayList, también existen las colecciones genéricas Stack<T>, Queue<T> y Dictionary<T>. Un ejemplo de la clase genérica de Stack seria:

```
Stack<int> pilaTexto = new Stack<int>();
```

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 seria 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 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.

También hay la posibilidad de crear nuestras propias clases y métodos genérico cuyo comportamiento estará ligado al tipo de dato que se utilice. Para lograr esto hay que hacer uso de los “type parameters”, los cuales permiten establecer un parámetro cuyo tipo se desconocerá hasta que se indique en la declaración.

```
class Impresora<T>{
	//atributos
	private string tipo;
	private Queue<T> cola;

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

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 se indica 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;
	}
}

public void imprimir(){
	Console.WriteLine();
	Console.WriteLine("Imprimiendo cola de tipo: {0}", tipo);
	Console.WriteLine();

	while(Cola.Count > 0){
		Console.WriteLine("{0} es de tipo : {1}", Cola.Dequeue(), tipo);
	}
}
```

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 la clase Impresora, lo cual indica que retornará 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, comprobar el funcionamiento de la clase Impresora:

```
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();
```

ENUMERACIONES

Una enumeración es un tipo especial de estructura en la que los literales de los valores que pueden tomar sus objetos se indican explícitamente en su definición. Por ejemplo, una enumeración de nombre Size cuyos objetos pudiesen tomar los valores literales Short, Medium o Large se definiría así:

```
public enum Size:int{
      Short = 9,
      Medium = 11,
      Large = 20
}
```

Estos tipos son muy útiles porque facilitan la escritura y legibilidad del código. Por ejemplo, en vez de definir un método que tome como parámetro un entero que indique el tamaño con el que se ha de mostrar un texto, es mejor definirlo con un parámetro de tipo Size ya que así el programador usuario no ha de recordar cual es el número y aparte es más fácil leer las llamadas al mismo.

```
Size size = Size.Medium;
Console.WriteLine(size); //salida Medium

int size2 =(int) Size.Medium;
Console.WriteLine(size2); //salida 11
```

Las restricciones de una enumeración son:

- No se pueden definir métodos.
- No se pueden implementar interfaces.
- No se pueden definir propiedades ni eventos.

El tipo por defecto de las constantes que forman una enumeración es int, aunque se puede dar cualquier otro tipo básico entero (byte, sbyte, short, ushort, uint, int, long o ulong) indicándolo en la enumeración.

La clase System.Enum

Todos los tipos enumerados derivan de System.Enum, que deriva de System.ValueType y ésta a su vez deriva de la clase primigenia System.Object. Aparte de los métodos heredados de estas clases padres, toda enumeración también dispone de otros métodos heredados de System.Enum, siendo los principales de ellos:

GetUnderlyingType(Type enum): Devuelve un objeto System.Type con información sobre el tipo base de la enumeración representada por el objeto System.Type que se le pasa como parámetro .

GetValues(Type enum): Devuelve un array con los valores de todos los literales de la enumeración representada por el objeto System.Type que se le pasa como parámetro.

GetName(Type enum, object valor): Devuelve una cadena con el nombre del literal de la enumeración representada por enum que tenga el valor especificado en valor. Si la enumeración no contiene ningún literal con ese valor devuelve null, y si tuviese varios con ese mismo valor devolvería sólo el nombre del último. Si se quiere obtener el de todos es mejor usar GetNames(), que se usa como GetName() pero devuelve un string[] con los nombres de todos los literales que tengan el valor indicado ordenados según su orden de definición en la enumeración.

isDefined (Type enum, object valor): Devuelve un booleano que indica si algún literal de la enumeración indicada tiene el valor indicado.

```
using System;

namespace ejemplo_26
{
	public enum Size:int{
		Short = 9,
		Medium = 11,
		Large = 20
	}

	class Program{
		public static void Main(){
			Object tipo = Enum.GetUnderlyingType(typeof(Size));
			Console.WriteLine(tipo); //salida System.Int32

			Array aValues= Enum.GetValues(typeof(Size));
			foreach(Size size in aValues)
				Console.WriteLine(size);

			string sizeName = Enum.GetName(typeof(Size), Size.Short);
			Console.WriteLine(sizeName);
		}
	}
}
```

Ejemplo usando estructuras y enumeraciones

```
using System;

namespace ejemplo_27
{
	public enum Sexo:int{
		Masculino,
		Femenino
	}

	public struct Alumno{
		public string nombre;
		public int semestre;
		public int edad;
		public Sexo sexo;

		public Alumno(string nombre, int semestre, int edad, Sexo sexo){
			this.nombre = nombre;
			this.semestre = semestre;
			this.edad = edad;
			this.sexo = sexo;
		}

		//Se sobrescribe metodo ToString que
		//permite imprimir el objeto de manera personalizada
		public override string ToString(){
			return nombre + " " + semestre + " " + edad + " " + sexo;
		}
	}

	class Program{
		static void Main(string[] args){
			Alumno alumno = new Alumno("Juan", 8, 19, Sexo.Masculino);
			Console.WriteLine(alumno.ToString());
		}
	}
}
```

ACTIVIDADES

1.- ¿Cuál es el índice del primer elemento de un array?

2.-¿Qué sucede si se intenta acceder al elemento n+1 de un array de n elementos?

3.- ¿Por qué motivo un array posee métodos y propiedades?

4.- ¿Puede una clase implementar simultáneamente las interfaces IEnumerable y ICollection?

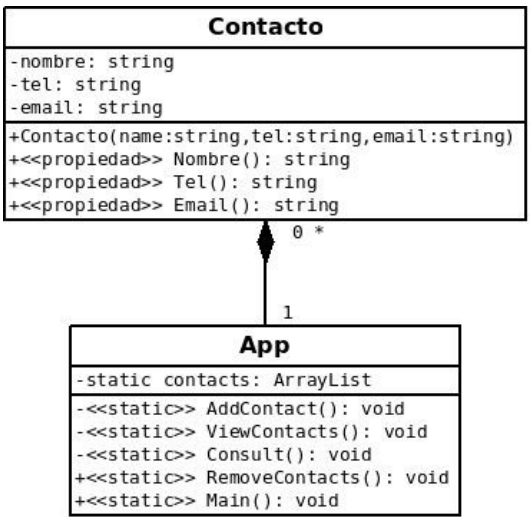
5.-Crear un programa de consola que permita principalmente:

- Ingresar el nombre, el teléfono y el email de personas (almacenar estos datos en un objeto de tipo ArrayList).
- Buscar contactos por medio del nombre.

Al ejecutar el programa aparece un menú con una serie de opciones:

1. Añadir contacto
2. Ver todos los contactos
3. Consultar
4. Borrar todos los contactos

- Usar un objeto que contenga los datos del contacto.(se puede usar una clase o una estructura)
- Al ver todos los contactos se muestra el indice, nombre, teléfono y email.
- Al consultar indicando un nombre aparecen los resultados encontrados que coinciden con el nombre mostrando los datos.
- En todos los pasos se puede volver al menú principal.



Pistas:

Para limpiar la consola se puede usar:

```
Console.Clear();
```

Para recoger lo escrito en la terminal:

```
string name = Console.ReadLine();
```

Para convertir una cadena en minúsculas:

```
name.ToLower()
```