



Lenguaje C#

Lenguaje Integrado de Consultas

LINQ



DISTRIBUIDO POR:

CENTRO DE INFORMÁTICA PROFESIONAL S.L.

C/ URGELL, 100
08011 BARCELONA
TFNO: 93 426 50 87

C/ RAFAELA YBARRA, 10
48014 BILBAO
TFNO: 94 448 31 33

www.cipsa.net

**RESERVADOS TODOS LOS DERECHOS. QUEDA PROHIBIDO TODO TIPO DE
REPRODUCCIÓN TOTAL O PARCIAL DE ESTE MANUAL, SIN PREVIO
CONSENTIMIENTO POR EL ESCRITOR DEL EDITOR**

1.- LINQ (Lenguaje Integrado de Consultas)

Una de las tareas más comunes en la programación es la realización de búsquedas en diferentes estructuras de datos tales como *matrices*, *colecciones*, *bases de datos*, *ficheros XML o binarios*... etc. Tradicionalmente las búsquedas de datos se han realizado programando pesados bucles de búsqueda o empleando lenguajes de consulta específicos, como por ejemplo SQL en las bases de datos.

Esta situación pasa a la historia con el lenguaje integrado de consultas o LINQ. Este lenguaje permite declarar consultas de manera natural contra cualquier estructura de datos como parte del código de cualquier aplicación de .NET. Sus características más destacables son:

- ⌚ Permite declarar búsquedas sobre cualquier estructura de datos en memoria. Sus principales ventajas son las siguientes:
- ⌚ Está integrado en .NET es utilizable dentro del propio código de una aplicación escrita tanto en C# como en Visual Basic. Esto no es posible con otros lenguajes de consultas como *SQL* para bases de datos o *XPath* en ficheros XML.
- ⌚ Es orientado a objetos y la consultas se definen en base a objetos que representan datos de igual modo que se manejan en las aplicaciones.

1.1.- Sintaxis básica de Consultas con LINQ

La estructura básica de una consulta sencilla en LINQ puede resumirse en:

```
IEnumerable<T> resultado =  
    from <registro> in <colección>  
    where <condiciones>  
    select <registro>
```

Donde *resultado* es un enumerador de objetos de la clase o estructura *T* que puede emplearse para recorrer el conjunto de objetos o valores resultantes de la consulta.

El elemento *colección* representa el conjunto de datos al que se aplica la consulta. El elemento *registro* es una referencia de tipo *T* que representa a cada objeto en la colección y permite el acceso a sus propiedades.

El elemento resultado de tipo *IEnumerable<T>* puede ser recorrido empleando un bucle *foreach* de tipo:

```
foreach( T valor in resultado ) {
```

```
    ...
```

```
}
```

Ejemplo de consulta simple con LINQ.

Sea la clase *Punto* que define el tipo de los objetos almacenados en la lista:

```
class Punto
{
    public int x;
    public int y;
    public string id;
    public Punto(int _x, int _y, string _id) {
        x = _x; y = _y; id = _id;
    }
    public override string ToString() {
        return "(" + x.ToString() + "," + y.ToString() + "," + id + ")";
    }
}
```

El siguiente código muestra la inicialización de una colección de tipo *List<Punto>* sobre la que se aplica una consulta en LINQ diseñada para devolver aquellos puntos cuyo atributo *id* vale "AB":

```
class Program
{
    static void Main(string[] args)
    {
        // Coleccion de puntos.
        List<Punto> puntos = new List<Punto>(); // Declaración y
        puntos.Add(new Punto(10, 20, "AB")); // Inicialización de
        puntos.Add(new Punto(11, 19, "CD")); // colección
        puntos.Add(new Punto(14, 16, "AB")); puntos.Add(new Punto(17, 13, "OP"));
        puntos.Add(new Punto(18, 12, "AB"));

        // Consulta que devuelve todos los objetos Punto de la coleccion
        // cuyo atributo identificador vale "AB"
        IEnumerable<Punto>
        puntosEnum = from punto in puntos where
        punto.id == "AB" select punto;

        // Bucle de recorrido de elementos en coleccion
        foreach (Punto p in puntosEnum) {
            Console.WriteLine(p.ToString());
        }
    }
}
```

La consulta devuelve un valor de tipo *IEnumerable<Punto>*, que permite recorrer el conjunto resultante objetos *Punto* cuya propiedad *id* vale "AB". Este puede recorrerse mediante un bucle *foreach* en el que cada elemento devuelto es un objeto de tipo *Punto*:

```
<10,20,AB>
<14,16,AB>
<18,12,AB>
```

1.2.- Cláusula origen de datos *From In*

Las cláusulas **From In** especifican el origen de datos para la consulta LINQ. Primero se coloca la variable de referencia *<variable>* (también llamada *variable de registro*) para

acceder a las propiedades de los objetos de la colección y después el origen de datos `<colección>`.

from <variable> in <colección>

El origen de datos puede ser cualquier colección de datos que implemente el interfaz ***IEnumerable<T>*** definido en el espacio de nombres ***System.Collections.Generic*** y que es propio de cualquier estructura de datos recorrible mediante un bucle *foreach*.

La variable de referencia es semejante a la variable de iteración que devuelve cada elemento de una colección en un bucle *foreach*. El tipo de esta variable es determinado automáticamente por el sistema en base al tipo de los objetos contenidos en el origen de datos.

En el ejemplo anterior, la colección puntos es de tipo *List<Punto>* la cual implementa el interfaz *IEnumerable<Punto>*; por lo tanto; la variable de referencia *punto* es de tipo *Punto*.

1.3.- Cláusula de filtrado *Where*

La cláusula ***Where*** permite determinar un criterio a través de una o varias condiciones que permita filtrar los objetos del conjunto de datos.

where <condición>

Normalmente `<condición>` comprende una o varias expresiones condiciones relacionadas con los valores de los atributos de la variable de referencia. Solo los objetos que cumplen las condiciones indicadas son incluidos en el conjunto de resultados de la consulta. En el ejemplo anterior:

```
where punto.id == "AB"
```

Restringe los objetos seleccionados a aquellos objetos cuyo atributo *id* valga "AB". Otros ejemplos de condiciones podrían ser:

```
where punto.id.StartsWith("M")
```

Filtra aquellos objetos cuyo atributo *id* comienza por M

```
where punto.x > 10
```

Retorna todos los objetos cuyo atributo *x* sea mayor que 10.

1.4.- Cláusula de proyección *Select*

La cláusula ***Select*** permite indicar qué devuelve la consulta con respecto a los objetos del origen de datos que han cumplido la condición indicada en la cláusula de filtrado *where*. En el caso más simple puede retornarse los propios objetos seleccionados indicando la propia variable de referencia:

```

        IEnumerable<Punto> puntosEnum =
        from punto in puntos
        where punto.x > 10
        select punto;
        // Bucle de visualización de objetos.
        foreach (Punto p in puntosEnum) {
        Console.WriteLine(p.ToString());
        }
    
```

Por otro lado, también puede retornarse el valor de un atributo de cada objeto. En el ejemplo, se retorna el valor del atributo id de cada objeto Punto:

```

        IEnumerable<String> puntosEnum =
        from punto in puntos
        where punto.x > 10
        select
        punto.id;

        // Bucle de visualización de identificaciones.
        foreach (String identificacion in puntosEnum)
        {
            Console.WriteLine(identificacion);
        }
    
```

En función del valor devuelto, cambia el tipo de los elementos que conforman el conjunto de resultados. Si se devuelven los propios objetos Punto, el resultado es enumerador de tipo Punto (*IEnumerable<Punto>*). Si se retornan los valores del atributo Id de cada objeto, el resultado es de tipo cadena (*IEnumerable<String>*)

1.5.- Cláusula de Unión *Join In*

En algunas ocasiones puede ser necesario realizar búsquedas sobre datos almacenados en diferentes orígenes de datos. LINQ introduce la cláusula ***join*** que permite combinar los datos de varios orígenes de datos.

From <var_ref1> In <origen_datos1>
Join >var_ref2> In <origen_datos2> On
<var_ref1>.propiedad Equals <var_ref2>.propiedad

Esta cláusula se incluye a continuación de las cláusulas ***From In*** de manera que se reciben dos orígenes de datos *<origen_datos1>* y *<origen_datos2>*, dos variables de referencia *<var_ref1>* y *<var_ref2>* y una expresión condicional *<condicion_join>* que determina la relación cada objeto del primer origen con uno del segundo.

La cláusula Join implementa lo que se denomina unión interna (*INNER JOIN*), de modo que sólo se evalúan los pares de objetos que pueden relacionarse aplicando la condición. LINQ no soporta uniones externas (*OUTER JOIN*).

Ejemplo: Sean las siguientes clases:

```
// Clase Almacen
public class Almacen
{
    public int id;
    public string zona;

    public Almacen(int _id, string _zona)
    {
        this.id = _id;
        this.zona = _zona;
    }
}

// Clase Producto
public class Producto
{
    public string nombre;
    public int stock;
    public int id_almacen;

    public Producto(string _nombre, int _stock, int _id_almacen)
    {
        this.nombre = _nombre;
        this.stock = _stock;
        this.id_almacen = _id_almacen;
    }
}
```

La clase *Almacen* permite representar objetos almacen con un identificador numérico y un descriptor de zona de tipo cadena. La clase *Producto* permite representar los diferentes productos almacenados.

Cada producto incluye un atributo *id_almacen* de tipo entero que se corresponde con el atributo *id* del objeto almacen en el que se encuentra.

Supóngase la inicialización de las siguientes colecciones genéricas **List**.

```
// Colección de almacenes
List<Almacen> almacenes = new List<Almacen>();
almacenes.AddRange(new Almacen[] { new Almacen( 0, "ALMACEN_0" ),
new Almacen( 1, "ALMACEN_1" ),
new Almacen( 2, "ALMACEN_2" ),
new Almacen( 3, "ALMACEN_3" )});

// Colección de productos.
List<Producto> productos = new List<Producto>();
productos.AddRange(new Producto[] { new Producto("PIPAS", 230, 0 ),
new Producto("CARAMELOS", 110, 0 ),
new Producto("GOLOSINAS", 10, 1 ),
new Producto("HELADOS", 210, 2 ),
new Producto("DULCES", 30, 2 ),
new Producto("ROSQUILLAS", 80, 1 ),
new Producto("TOSTADA", 80, 4 ),
new Producto("BOMBONES", 20, 0 )});
```

Deseamos realizar una consulta que devuelva aquellos productos cuyo Stock sea inferior a 100 unidades junto con el nombre del almacén en el que están. Esta consulta va a

retornar diferentes valores en vez de uno sólo y eso plantea un problema ya que el conjunto de resultados debe ser de un tipo iterador *IEnumerable<T>*, que únicamente permite definir un único tipo de dato.

Una posible solución es definir un tipo de dato específico que contenga todos los valores que desean retornarse. Si se desea obtener por cada producto su nombre, su stock, y el nombre del almacén al que pertenece; debería crearse una clase:

```
class AlmacenProducto
{
    public string zonaAlmacen;
    public string nombreProducto;
    public int stock;
}
```

La consulta podría expresarse mediante LINQ con el siguiente código:

```
// Ejecución de consulta
IEnumerable<AlmacenProducto> resultado =
from p in productos join a in
almacenes on p.id_almacen equals a.id
where p.stock < 100
select new AlmacenProducto
{
    zonaAlmacen = a.zona,
    nombreProducto = p.nombre, stock
= p.stock
};

// Bucle de visualización
foreach (AlmacenProducto valor in resultado)
{
    Console.WriteLine("{0} {1} {2}",
        valor.zonaAlmacen,
        valor.nombreProducto,
        valor.stock);
}
```

El tipo del conjunto de resultados es *IEnumerable<AlmacenProducto>* para dar cabida a los valores que desean retornarse.

El resultado mostrado por pantalla es:

```
ALMACEN_1 GOLOSINAS 10
ALMACEN_2 DULCES 30
ALMACEN_1 ROSQUILLAS 80
ALMACEN_0 BOMBONES 20
```

Resultados de la consulta

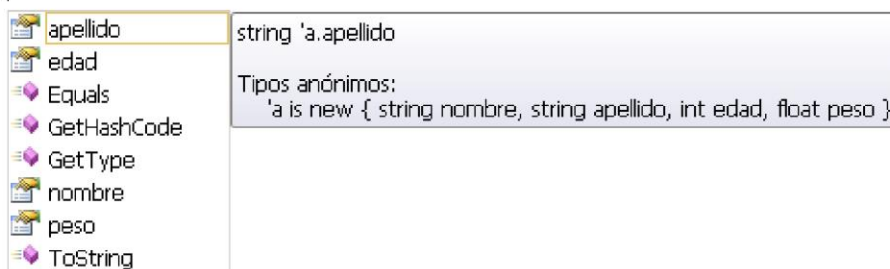
Aquellos objetos que no cumplen la condición de unión, es decir; productos cuyo atributo *id_zona* no coincide con el valor *id* de ningún almacén, así como almacenes cuyo

Ejemplo, el producto TOSTADAS tiene 80 de Stock y debería aparecer en los resultados puesto que está por debajo de las 100 unidades. Sin embargo, su atributo *id_zona* = 4 y no existe ningún objeto almacen cuyo *id* = 4, por lo que no puede emparejarse a ningún objeto almacen y es excluido.

1.6.- Tipos anónimos y variables de tipo implícito.

Definir una clase para poder almacenar un determinado conjunto de valores como resultado de una consulta LINQ puede resultar pesado. Para evitarlo se puede emplear un **tipo anónimo**. Un tipo anónimo permite la creación de objetos de una clase que no existe. Ejemplo:

```
// Declaracion del objeto persona de tipo anónimo.
var persona = new { nombre = "Roger", apellido = "Petroviano", edad = 23, peso = 82.5f };
persona.
```



atributo *id* no se refleja en el atributo *id_almacen* de ningún producto existente no son considerados ni aparecen en los resultados.

La sentencia crea un objeto de una clase *anónima* que ni tiene nombre ni puede instanciarse más veces. Esta clase se crea en ejecución el instanciarse el objeto a partir de los valores que se indican. En este caso, el tipo anónimo asociado al objeto creado posee tres atributos (*nombre, apellido, edad y peso*).

Dado que el tipo del objeto no está definido; la variable de instancia *persona* se declara de **tipo implícito**. Una variable de tipo implícito es aquella que se declara sin indicar su tipo y es el compilador el que lo establece en ejecución según el valor que se le asigna.

Las variables de tipo implícito se declaran anteponiendo a su identificador la palabra clave **var**:

```
var id = 0; // id de tipo int
var nombre = "Yuri"; // nombre de tipo string.
var personas = new List<Almacen>(); // personas de tipo Lista de Persona.
var posicion = new { x = 10, y = 20, z = 30 }; // posicion de tipo anónimo.
```

Mediante el uso de una variable de tipo implícito y un tipo anónimo es posible reescribir el código de la consulta anterior sin necesidad de crear la clase *AlmacenProducto*:

```
// Ejecución de consulta
var resultado = from
p in productos
join a in almacen on p.id_zona equals a.id
where p.stock < 100
select new
{
    zonaAlmacen = a.zona,
    nombreProducto = p.nombre,
    stock = p.stock
};

// Visualización de resultados.
foreach( var objeto in resultado ) {
    Console.WriteLine("{0} {1} {2}",
        objeto.nombreProducto,
        objeto.stock,
        objeto.zonaAlmacen,
    );
}
```

El resultado es un iterador de tipo anónimo que reúne algunos atributos de los objetos Almacen y Producto cuyos atributos *id* e *id_zona* son coincidentes.

1.7.- Cláusula de Agrupamientos *Group By Into*

La cláusula **Group** permite agrupar los objetos de un origen de datos en subgrupos en función del valor de uno de sus atributos (*propiedad_clave*). Los grupos resultantes se almacenen en una variable de referencia nueva (*var_ref_agrupamiento*).

```
group <var_ref>
by <var_ref>.<propiedad_clave>
into <var_ref_agrupamiento>
```

La cláusula **Group By** genera al ejecutarse un conjunto de resultados asociado a la variable de referencia indicada por la cláusula **Into** (*var_ref_agrupamiento*). Cada objeto de este conjunto de resultados contiene una propiedad *Key* con un valor de agrupamiento, y otro subiterador para recorrer todos los objetos agrupados bajo el valor indicado por la propiedad *Key*.

Ejemplo: Sea la colección de productos del ejemplo anterior, supóngase que se desea realizar una consulta que retorne todos los productos agrupados por almacén.

```
// Colección de productos.
List<Producto> productos = new List<Producto>();
productos.AddRange(new Producto[] { new Producto("PIPAS", 230, 0 ),
new Producto("CAMELOS", 110, 0 ),
new Producto("GOLOSINAS", 10, 1 ),
new Producto("HELADOS", 210, 2 ),
new Producto("DULCES", 30, 2 ),
new Producto("ROSQUILLAS", 80, 1 ),
new Producto("TOSTADA", 80, 4 ),
new Producto("BOMBONES", 20, 0 )});

// Consulta
var resultado =
in productos
p.id_almacen
grupoProductos
from p
group p by
into
select new { idAlmacen = grupoProductos.Key, productos
= grupoProductos };
```

La consulta devuelve un conjunto de objetos con dos atributos: *idAlmacen* que se corresponde con la identificación de un almacén, y *productos* que contiene un subconjunto con todos los objetos *Producto* asociados al almacén.

```
// Visualización de resultados.
foreach (var resultado in resultados)
{
    // Muestra id_zona del grupo
    Console.WriteLine("GRUPO ID ZONA -> " + resultado.idAlmacen);
    // Bucle de recorrido de subiterador de grupo
    foreach (Producto producto in resultado.productos)
    {
        // Muestra objetos producto de cada id_zona
        Console.WriteLine("\t PRODUCTO -> " + producto.nombre);
    }
}
```

El resultado mostrado será:

```
GRUPO ID ZONA -> 0
PRODUCTO -> PIPAS
PRODUCTO -> CAMELOS
PRODUCTO -> BOMBONES
GRUPO ID ZONA -> 1
PRODUCTO -> GOLOSINAS
PRODUCTO -> ROSQUILLAS
GRUPO ID ZONA -> 2
PRODUCTO -> HELADOS
PRODUCTO -> DULCES
GRUPO ID ZONA -> 4
PRODUCTO -> TOSTADA
```

También es posible realizar agrupamientos en operaciones de unión empleando datos procedentes de dos orígenes de datos relacionados. Para ello se emplea la cláusula de unión ya vista (*Join In*) añadiéndole la cláusula **Into**.

Ejemplo: A partir de las clases *Almacen* y *Producto* de anteriores ejemplos, se desea obtener todos los productos registrados agrupados por almacen indicando la zona de cada almacen, y el nombre y stock de cada producto:

```
// Colección de almacenes
List<Almacen> almacenes = new List<Almacen>();
almacenes.AddRange(new Almacen[] { new Almacen( 0, "ALMACEN_0" ),
new Almacen( 1, "ALMACEN_1" ),
new Almacen( 2, "ALMACEN_2" ),
Almacen( 3, "ALMACEN_3" )});
// Colección de productos.
List<Producto> productos = new List<Producto>();
productos.AddRange(new Producto[] { new Producto("PIPAS", 230, 0 ),
new Producto("CARAMELOS", 110, 0 ),
new Producto("GOLOSINAS", 10, 1 ),
new Producto("HELADOS", 210, 2 ),
new Producto("DULCES", 30, 2 ),
new Producto("ROSQUILLAS", 80, 1 ),
new Producto("TOSTADA", 80, 4 ),
new Producto("BOMBONES", 20, 0 )});
// Consulta
var
resultados =
almacenes
join p in
productos
on a.id equals
p.id_almacen
into
grupoProductos

select new { almacen = a, productos = grupoProductos };
```

La consulta LINQ del ejemplo realiza una unión entre los objetos de las listas *almacenes* y *productos* según el valor de sus atributos *id* y *id_almacen*.

Dado que varios productos pueden hacer referencia al mismo almacen (si tienen el mismo valor en su atributo *id_almacen*), pueden agruparse en conjuntos por cada objeto almacen de la lista *almacenes*. Esos subconjuntos de resultados se establecen en la variable de referencia: *grupoProductos*.

El conjunto de resultados final consta de objetos con dos atributos: *almacen*, que hace referencia a cada objeto almacen presente en la colección *almacenes*, y *productos*, que representa un subconjunto de resultados con todos los objetos almacen asociados al almacen.

El código para la visualización de resultados sería el siguiente:

```
// Visualización de resultados
foreach (var resultado in resultados)
{
    Console.WriteLine("Almacen : {0}", resultado.almacen.zona);
    foreach (Producto producto in resultado.productos)
    {
        Console.WriteLine("\t Producto: {0} Stock: {1}",
            producto.nombre, producto.stock);
    }
}
```

Los valores mostrados son:

```

Almacen : ALMACEN_0
          Producto: PIPAS Stock: 230
          Producto: CARAMELOS Stock: 110
          Producto: BOMBONES Stock: 20
Almacen : ALMACEN_1
          Producto: GOLOSINAS Stock: 10
          Producto: ROSQUILLAS Stock: 80
Almacen : ALMACEN_2
          Producto: HELADOS Stock: 210
          Producto: DULCES Stock: 30
Almacen : ALMACEN_3

```

1.8.- Cláusula de Ordenación *OrderBy*

Esta cláusula permite indicar el orden en el que los resultados deben ordenarse. Para ello se emplea el valor de uno o varios atributos para ordenar los elementos del conjunto de resultados en orden ascendente o descendente.

OrderBy <var_ref1>.<prop1>,<var_ref1>.<prop2>.... <ascending>/<descending>

Los objetos de la lista de resultados se ordenan en función de los valores de los campos indicados en el orden en que se indican, tal que si dos objetos poseen el mismo valor para el primer atributo, se emplea los valores del segundo. Las cláusulas *ascending* y *descending* determinan el sentido de la ordenación.

Ejemplo: Supongamos dos listas con objetos de las clases Almacen y Productos definidas en los ejemplos anteriores:

```

// Colección de almacenes
List<Almacen> almacenes = new List<Almacen>();
almacenes.AddRange(new Almacen[] { new Almacen( 0, "ALMACEN_0" ),
                                     new Almacen( 1, "ALMACEN_1" ),
                                     new Almacen( 2, "ALMACEN_2" ),
                                     new Almacen( 3, "ALMACEN_3" )});

// Colección de productos.
List<Producto> productos = new List<Producto>();
productos.AddRange(new Producto[] { new Producto("PIPAS", 230, 0 ),
                                     new Producto("CARAMELOS", 110, 0 ),
                                     new Producto("GOLOSINAS", 10, 1 ),
                                     new Producto("HELADOS", 210, 2 ),
                                     new Producto("DULCES", 30, 2 ),
                                     new Producto("ROSQUILLAS", 80, 1 ),
                                     new Producto("TOSTADA", 80, 4 ),
                                     new Producto("BOMBONES", 20, 0 )});

```

Ejemplo: La siguiente consulta devuelve una lista con el nombre del producto, stock y el nombre del almacén, ordenados por nombre de producto:

```

// Consulta
var resultado =
from p in productos
join a in almacenes on p.id_almacen equals a.id
orderby p.nombre ascending
select new
{
    zonaAlmacen = a.zona,
    nombreProducto = p.nombre,
    stock = p.stock
};

// Visualización
foreach (var valor in resultado)
{

```

```

Console.WriteLine("{0} {1} {2}",
    valor.nombreProducto, valor.stock, valor.zonaAlmacen);
}

```

El resultado mostrado es:

```

BOMBONES 20 ALMACEN_0
CARAMELOS 110 ALMACEN_0
DULCES 30 ALMACEN_2
GOLOSINAS 10 ALMACEN_1
HELADOS 210 ALMACEN_2
PIPAS 230 ALMACEN_0
ROSQUILLAS 80 ALMACEN_1

```

Ejemplo2: La siguiente consulta retorna la misma lista de resultados pero ordenador por el identificador de almacén y el nombre de producto. El objetivo es que los resultados aparezcan ordenados por almacén, y a su vez, los productos del mismo almacén aparezcan ordenados por su nombre.

```

// Consulta
var resultado =
    from p in productos
    join a in almacen on p.id_almacen equals a.id
    orderby a.id, p.nombre ascending
    select new {
        zonaAlmacen = a.zona,
        nombreProducto = p.nombre,
        stock = p.stock
    };

```

```

// Visualización
foreach (var valor in resultado)
{
    Console.WriteLine("{0} {1} {2}",
        valor.zonaAlmacen, valor.nombreProducto, valor.stock);
}

```


El resultado mostrado es:

```
ALMACEN_0 BOMBONES 20
ALMACEN_0 CARAMELOS 110
ALMACEN_0 PIPAS 230
ALMACEN_1 GOLOSINAS 10
ALMACEN_1 ROSQUILLAS 80
ALMACEN_2 DULCES 30
ALMACEN_2 HELADOS 210
```

2.- Métodos Extendidos

2.1.- Definición de métodos extendidos.

Los métodos extendidos son un tipo especial de métodos que pueden añadirse a clases ya definidas tal y como si hubiesen sido definidos como parte de ellas originalmente.

Estos métodos pueden añadirse a cualquier clase incluidas las del framework de .NET para añadir más funcionalidades. Los métodos de extensión pueden emplearse como alternativa a la creación de clases hijas para añadir o sobrescribir los métodos de la clase padre.

Los métodos extendidos pueden ser tanto procedimientos como funciones y deben declararse estáticos dentro de una clase también estática. Al declararlos, el primer parámetro debe ser del tipo de la clase a la que extiende el método precedido de la partícula **this**. Por ejemplo; un método extendido para la clase String requiere la siguiente declaración:

```
public static <tipo_retorno> Identificador(this string objOrigen, ....)
{
    ... código
}
```

El parámetro de entrada **this String objOrigen** hace referencia al propio objeto cadena contra el que se llama al método de extensión, y equivale a **this** si el método estuviese definido dentro de la propia clase **String**.

Ejemplo: Supóngase que se desea crear un método extendido para la clase *String* denominado *Left* que retorne una cadena con el número de caracteres a la izquierda indicados por parámetro:

```
// Clase estática contenedora para método de extensión
public static class Libreria
{
    /// <summary>
    /// Retorna la cantidad de caracteres indicada del principio de la cadena
    /// </summary>
```

```

/// <param name="cadena">Objeto origen</param>
/// <param name="ncars">Cantidad de caracteres a devolver</param>
/// <returns>Subcadena con los caracteres indicados al principio de la cadena de
origen</returns>
public static string Left(this string cadena, int ncars)
{
    if (ncars < 0 || ncars > cadena.Length)
        return cadena;
    else
        return cadena.Substring(0, ncars);
}

```

La llamada al método de extensión se codifica del mismo modo que un método propio de la clase. El método extendido es reconocido a todos los efectos como método de la clase String.

```

String persona = "Roger";
String cadena = persona.Le

```




Al invocar al método, el primer parámetro no aparece en la llamada, sólo se dan valores para el resto:

```

String persona = "Roger";
String cadena = persona.Left(2);
Console.WriteLine(cadena);

```



Los métodos extendidos poseen dos limitaciones a tener presentes en su implementación:

- Un método de extensión sólo puede acceder a los métodos y atributos públicos del tipo al que se asocia. No se permite el acceso a miembros privados o protegidos de la misma.
- Si un método extensión entra en conflicto con un método propio de la clase, prevalece el método originario.

3.- Métodos Extendidos de LINQ

La forma de codificar LINQ vista es muy semejante a SQL. SQL es un lenguaje de consultas no orientado a objetos que se diferencia en gran medida de C#. Sin embargo, LINQ es ejecutado íntegramente con C#, que es orientado a objetos y funciona mediante invocaciones a métodos definidos en clases o estructuras. Por ello, la forma de expresar las consultas en LINQ puede parecer incompatible con C#.

Para que LINQ pueda ejecutarse íntegramente con C# es convertido internamente en llamadas a métodos extendidos asociados al interfaz *IEnumerable<T>*. Estos métodos están presentes a su vez en múltiples clases hijas que implementan colecciones como por ejemplo *List<T>* de *System.Collections.Generics*.

Estos métodos extendidos están definidos en la clase estática *System.Linq.Enumerable* y pueden ser invocados manualmente como alternativa al uso de LINQ si no se está familiarizado con la sintaxis estilo SQL.

Ejemplo: La siguiente consulta devuelve todos los productos presentes en el almacén con id = 0.

```
// Colección de productos.
List<Producto> productos = new List<Producto>();
productos.AddRange(new Producto[] { new Producto("PIPAS", 230, 0 ),
new Producto("CAMELOS", 110, 0 ),
new Producto("GOLOSINAS", 10, 1 ),
new Producto("HELADOS", 210, 2 ),
new Producto("DULCES", 30, 2 ),
new Producto("ROSQUILLAS", 80, 1 ),
new Producto("TOSTADA", 80, 4 ),
new Producto("BOMBONES", 20, 0 )});
IEnumerable<Producto> resultado = from p in productos
where p.id_almacen == 0
select p;
```

La misma consulta puede expresarse empleando el método extendido **Where** que puede invocarse sobre la colección productos de tipo *List<Producto>*:

```
IEnumerable<Producto> resultado2 = productos.Where(p=>(p.id_almacen == 0));
```

El método **Where** es un método extendido que devuelve un enumerador con todos los objetos contenidos en la colección producto que verifican el delegado *predicate* que se requiere como parámetro:

```
public static IEnumerable<TSource> Where<TSource>( this
    IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

El delegado exige una función que recibe un objeto del tipo correspondiente a los objetos presentes en la colección y el retorno de un valor booleano indicando si cumple o no la condición de filtrado.

A continuación se presentan los principales métodos extendidos definidos en *System.Linq.Enumerable*. La lista completa de los métodos de extensión definidos en esta clase puede consultarse en el sitio web del MSDN:

<http://msdn.microsoft.com/es-es/library/system.linq.enumerable.aspx>

3.1.- Método Extendido de filtrado *Where*.

Retorna un iterador a una colección generada a partir de los elementos de otra que cumplen una determinada condición indicada.

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

El método requiere una expresión lambda que recibe cada objeto de la colección y retorna un valor booleano indicando si cumple o no la condición de filtrado.

Ejemplo: La siguiente consulta devuelve todos los objetos de la colección productos cuyo stock está por debajo de las 10 unidades:

```
IEnumerable<Producto> resultados = productos.Where(p => p.stock < 10);
```

3.2.- Método Extendido de proyección *Select*

Retorna un iterador a una colección generada a partir de elementos de objetos de otra permitiendo crear un tipo anónimo que contenga en sus campos los valores de determinados atributos de los objetos de la colección original.

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source, Func<TSource,
    TResult> selector);
```

El método requiere una expresión lambda que recibe cada objeto de la colección y devuelve un valor, u objeto de tipo anónimo con los valores que se indiquen.

Ejemplo: La siguiente consulta devuelve el nombre de todos los productos cuyo stock es superior a 10 unidades:

```
IEnumerable<String> resultados = productos.Where(p => p.stock < 10)
    .Select( p => p.nombre );
```

El método *Select* puede concatenarse al método *Where*. Al hacerlo el método *Select* realiza la proyección sobre el enumerador con los objetos filtrados por el método *Where*.

- Método Extendido de

Retorna

de

3.3. unión *Join*.

un iterador a una colección generada a partir de cada par de elementos de otras dos colecciones que poseen determinados campos con el mismo valor.

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector);
```

El método requiere cuatro elementos:

- La colección con la que se realiza la unión
- Una expresión lambda que retorna la propiedad por la que se combinan los objetos de la colección contra la que se invoca el método JOIN.
- Una expresión lambda que retorna la propiedad por la que se combinan los objetos de la colección con la que se realiza la unión.
- Una expresión lambda que retorna el tipo anónimo que combina los atributos deseados de los objetos de ambas colecciones.

Ejemplo: La siguiente consulta realiza una unión de los objetos de la colección *productos* con los objetos de la colección *almacenes*. Los objetos de ambas colecciones se asocian por los atributo *id_almacen* de los *productos* (*p => p.id_almacen*) e *id* de los *almacenes* (*a => a.id*). El resultado de la unión es una tipo anónimo que contiene por cada par de objetos almacen-producto; el nombre, stock y zona de almacen. (*p,a*) => *new { nombre = p.nombre, stock = p.stock, almacen = a.zona }*

```
var resultados = productos.Join( almacenos,
    p => p.id_almacen,
    a => a.id,
    ( p, a ) => new { nombre = p.nombre,
                    stock = p.stock,
                    almacen = a.zona });

foreach (var obj in resultados)
{
    Console.WriteLine("PRODUCTO {0} STOCK {1} ALMACEN ZONA {2}",
        obj.nombre, obj.stock, obj.almacen);
}
```

3.4.- agrupación *GroupBy*

Retorna una enumerador con todos los valores de un atributo indicado como clave de agrupamiento, encontrado en los objetos de la colección original. Cada objeto del

Método Extendido de

de resultado lleva asociado un enumerador anidado con todos los elementos de la colección original que poseen el mismo valor en el atributo de agrupamiento.

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
```

El método requiere de una expresión lambda que recibe cada objeto de la colección e indica el atributo por el que desean agruparse los objetos.

Ejemplo 1: La siguiente consulta devuelve los productos almacenados en la colección productos agrupados por el id_almacen.

```
var resultados = productos.GroupBy(p => p.id_almacen).
    Select( r => new {
        IdAlmacen = r.Key,
        Productos = r
    }
);
```

El resultado es un objeto de tipo *IGruoping* que hereda de *IEnumerable*. Cada objeto contiene un valor Key con el valor de agrupamiento y un enumerador con todos los objetos agrupados por dicho valor:

```
foreach( var valor in resultados ) {
    Console.WriteLine("ID_ALMACEN: {0}", valor.IdAlmacen);
    foreach (Producto p in valor.Productos)
    {
        Console.WriteLine("NOMBRE {0} STOCK {1}", p.nombre, p.stock);
    }
}
```

En este caso, el valor de agrupamiento es el id_almacen de cada producto. El bucle externo recorre todas las claves (que se corresponden con los diferentes valor id_almacen encontrados en los objetos producto), y el bucle anidado recorre el conjunto de objetos producto asociados a cada valor clave.

El resultado es el siguiente:

```
ID_ALMACEN: 0
    NOMBRE PIPAS STOCK 230
    NOMBRE CARAMELOS STOCK 110
    NOMBRE BOMBONES STOCK 20
ID_ALMACEN: 1
    NOMBRE GOLOSINAS STOCK 10
    NOMBRE ROSQUILLAS STOCK 80
ID_ALMACEN: 2
    NOMBRE HELADOS STOCK 210
    NOMBRE DULCES STOCK 30
ID_ALMACEN: 4
    NOMBRE TOSTADA STOCK 80
```

- Método Extendido de

Retorna

de

3.5. ordenación *OrderBy* / *OrderByDecreasing*

un enumerador con todos los objetos ordenados en función de los valores un determinado atributo.

```
public static IEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
```

El método requiere una expresión lambda que recibe cada objeto de la colección e indica el atributo por el que se quieren reordenar.

Método Extendido de

de

Ejemplo: La siguiente consulta devuelven todos los objetos de la colección productos ordenados por stock:

```
var ProductosOrdenados = productos.OrderBy( p => p.stock );
```

En este caso el resultado son todos los objetos contenidos en la colección productos ordenados por stock de menor a mayor:

```
foreach ( Producto p in ProductosOrdenados ) {
    Console.WriteLine("STOCK {0} PRODUCTO {1}", p.stock, p.nombre );
}
```

El resultado será:

```
STOCK 10 PRODUCTO GOLOSINAS
STOCK 20 PRODUCTO BOMBONES
STOCK 30 PRODUCTO DULCES
STOCK 80 PRODUCTO ROSQUILLAS
STOCK 80 PRODUCTO TOSTADA
STOCK 110 PRODUCTO CARAMELOS
STOCK 210 PRODUCTO HELADOS
STOCK 230 PRODUCTO PIPAS
```

3.6.- Métodos Extendidos de Agregados.

- **Count** ➤ El método devuelve el número de objetos que cumplen la condición indicada mediante una expresión lambda. El valor retornado es de tipo entero.

```
// Contabiliza todos los productos cuyo stock es menor de 10 unidades
int valor = productos.Count(p => p.stock < 10);
```

- **Sum** ➤ El método devuelve el sumatorio de los valores de una propiedad indicada mediante una expresión lambda en los objetos de la colección. El valor retornado depende del tipo de la propiedad seleccionada.

```
// Retorna el sumatorio del stock de todos productos.
int valor = productos.Sum(p => p.stock);
```


-

- **Average** ➤ El método devuelve la media de los valores de una propiedad indicada mediante una expresión lambda en los objetos de la colección. El valor retornado es de tipo *Double*.

```
// Retorna la media de stock de todos los productos en la coleccion
double valor = productos.Average(p => p.stock);
```

- **Max, Min** ➤ Devuelve el valor máximo de una propiedad de los objetos de una colección.

```
// Retorna el máximo y el mínimo del Stock de todos los productos
int maximoStock = productos.Max(p => p.stock);      int minimoStock =
productos.Min(p => p.stock);
```

- **All, Any** ➤ Devuelven un valor lógico cierto si alguno de los objetos de una colección, o todos ellos; verifican una determinada condición.

Ejemplo: La siguiente consulta devuelve un valor lógico cierto, sí y solo sí, todos los objetos de la colección productos tienen un stock > 0.

```
bool StocksOk = productos.All(p => p.stock > 0);
```

Ejemplo: La siguiente consulta devuelve un valor lógico cierto, sí y solo sí, algún objeto de la colección productos tienen un stock = 0.

```
bool PedirProductos = productos.Any(p => p.stock == 0);
```

3.7.- Combinación de métodos extendidos.

La invocación en secuencia de todos estos métodos uno detrás del otro es posible debido a que cada método es miembro extendido de *IEnumerable<T>* y retorna en su mayoría a su vez una instancia *IEnumerable<T>*. Sin embargo, si intentamos localizar la definición de estos métodos en dicho interfaz no la encontraremos.

Recordar que los métodos vistos (*join, where, select, orderby...*), son métodos extendidos añadidos al interfaz *IEnumerable<T>* y no pertenecen a su definición original. Todos ellos están definidos en la clase estática *System.Linq.Enumerable*. Para su uso debe importarse primero el espacio de nombres *System.Linq*.

3.8. Resumen: Expresiones Lambda en Métodos Extendidos de LINQ.

La mayoría de los métodos extendidos de LINQ mostrados anteriormente poseen un *predicado* como parámetro: Ejemplo:

Func<T1, T2> funcion

Un *predicado* es un delegado a una función que evalúa una determinada condición o expresión en base a un valor de entrada de un determinado tipo y uno de retorno de un tipo también determinado. Ejemplo:

Sea la estructura:

```
class Area
{
    public String id;
    public int x;
    public int y;
    public double r;

    public Area(String _id, int _x, int _y, double _r)
    {
        id = _id; x = _x; y = _y; r = _r;
    }
}
```

Supongamos que tenemos la siguiente matriz de valores Area:

```
// Colección de areas.
Area[] areas = new Area[] { new Area("AREA01", 12,23, 0.4),
new Area("AREA02", 23,11, 0.6),
new Area("AREA03", 22,45, 0.8),
new Area("AREA04", 33,45, 0.6)};
```

El método **Where** define un parámetro *predicate* que es un delegado para una función que recibe un objeto de la colección contra la que se ejecuta la consulta, y devuelve un valor de tipo booleano que determina si cumple o no la condición de filtrado.

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

Es posible invocarlo empleando un método anónimo.

```
IEnumerable<Area> resultado = areas.Where(delegate(Area e) { return e.id == "0000"; });
```


-

Otra alternativa es el empleo de una *expresión lambda* que se ajuste a las especificaciones del delegado. Las expresiones lambdas son expresiones funcionales que siguen el siguiente diseño:

<parámetro_entrada> => <expresión_lambda>

El parámetro de la izquierda representa la entrada y la derecha la expresión lambda responsable de generar el valor de retorno. Por ejemplo, el delegado en línea del anterior ejemplo podría implementarse mediante la siguiente expresión lambda:

```
IEnumerable<Area> resultado = areas.where(area => area.id == "000000");
```

Donde *area* a la parte izquierda de la expresión es un mero identificador que representa cada objeto de la colección *areas*, y a la parte derecha se indica la condición de filtrado.

El método **Select** define un parámetro selector que es un delegado a una función de selección que reciben un tipo y devuelven otro.

```
public static IEnumerable<TResult> Select<TSource, TResult>(  
    this IEnumerable<TSource> source,  
    Func<TSource, TResult> selector  
)
```

Es posible invocarlo mediante un método anónimo:

```
IEnumerable<String> idAreas = areas.Select(delegate(Area a) { return a.id; });
```

Mediante expresión lambda:

```
IEnumerable<String> idAreas2 = areas.Select(area => area.id);
```

En el caso de que deseemos devolver los valores de dos atributos de cada objeto sería necesario definir un nuevo tipo contenedor, o un tipo anónimo mediante la palabra clave **var**:

```
// Consulta  
var resultado = areas.Select(area => new { coordX = area.x, coordY = area.y  
});  
// visualizacion  
foreach (var objeto in resultado)  
{  
    Console.WriteLine("X: {0} Y: {1}", objeto.coordX, objeto.coordY);  
}
```

En este caso, la variable *resultado* representa un iterador de una colección de elementos de un tipo anónimo provisto de los atributos *coordX* y *coordY*.

3.9. Evacuación diferida en LINQ

Las consultas LINQ no generan una colección de resultados independientes del origen de datos. El resultado es en realidad un objeto enumerador *IEnumerable<T>* que permite recorrer un subconjunto de los objetos de la colección. Cualquier modificación sobre la colección de origen (inclusión, eliminación de objetos nuevos) se refleja automáticamente en el conjunto de resultados puesto que se recalculan cuando se recorre.

-

Ejemplo:

```

class Punto
{
    public int x;
    public int y;
    public string id;
    public Punto(int _x, int _y, string _id) {
        x = _x; y = _y; id = _id;
    }
    public override string ToString() {
        return "(" + x.ToString() + "," + y.ToString() + "," + id + ")";
    }
}

```

Si ejecutamos el siguiente código:

```

List<Punto> puntos = new List<Punto>(); // Declaración y
puntos.Add(new Punto(10, 20, "AB")); // Inicialización de
puntos.Add(new Punto(11, 19, "CD")); // colección
puntos.Add(new Punto(14, 16, "AB")); puntos.Add(new Punto(17, 13,
"OP")); puntos.Add(new Punto(18, 12, "AB"));

// Obtengo enumerador de resultados. Todos aquellos cuyo campo ID =
"AB" IEnumerable<Punto> puntosEnum = from punto in
puntos select punto; // Bucle de recorrido de resultados.
Console.WriteLine("LISTADO 1"); foreach (Punto p
in puntosEnum) { Console.WriteLine("\t" +
p.ToString()); }
// Modificación de la colección.
puntos[1] = new Punto(0,0,"AB");

// Bucle de recorrido de elementos en colección
Console.WriteLine();
Console.WriteLine("LISTADO 2");
foreach (Punto p in puntosEnum) {
    Console.WriteLine("\t" + p.ToString());
}

```

```

LISTADO 1
(10,20,AB)
(14,16,AB)
(18,12,AB)

LISTADO 2
(10,20,AB)
(0,0,AB)
(14,16,AB)
(18,12,AB)

```

La consulta del código devuelve un conjunto de resultados *puntosEnum*. En la primera visualización se muestran todos los objetos Punto cuyo atributo ID = "AB".

A continuación se inserta un nuevo Punto en la colección origen con el atributo ID = "AB". Al recorrer por segunda vez el mismo conjunto de resultados aparece el nuevo objeto Punto insertado.

La evaluación diferida de LINQ permite por lo tanto obtener los resultados más recientes cada vez que se recorre el subconjunto de resultados.

No obstante; en algunas situaciones puede ser deseable obtener una copia independiente con todos los objetos resultantes de la consulta. Para ello pueden emplearse los métodos ***ToArray<T>()*** o ***ToList<T>()*** del enumerador resultante, donde T representa la clase o estructura a la que pertenece cada objeto o valor resultado.

```
// Obtengo enumerador de resultados. Todos aquellos cuyo campo ID = "AB"
IEnumerable<Punto> puntosEnum =           from punto in puntos
where punto.id == "AB"                   select punto;

// Matriz de objetos Punto.
Punto[] datos = puntosEnum.ToArray<Punto>();
```

El método ***ToArray()*** retorna los objetos que conforman el conjunto de resultados en la forma de una matriz ***T[]***. El método ***ToList()*** es equivalente, pero retorna los objetos contenidos en una colección generica tipo ***List<T>***.

Ejercicios

La tienda

Modifica el ejercicio “La tienda” del anexo de colecciones realizado anteriormente en modo consola.

- ⌚ Añade un interfaz gráfico mediante ventanas que permita realizar las operaciones requeridas por la aplicación. Preferentemente emplea un interfaz de ventanas MDI.
- ⌚ Modifica las operaciones que implican búsquedas y ordenamientos en las clases *ColecciónArticulos*, *ColecciónCliente* y *ColecciónAlquileres*, empleando ahora LINQ en vez de bucles para cada operación.
- ⌚ Añade una clase *GestionFichero* que permite cargar y guardar en ficheros el conjunto de clientes, artículos y alquileres almacenados. La clase debe constar de los siguientes métodos:

```
public class GestionFichero
{
    public GestionFichero(String ruta);

    public void GuardarArticulos(Articulo[] coleccion);

    public void GuardarClientes(Cliente[] coleccion);

    public void GuardarAlquiler(Alquiler[] coleccion);

    public Articulo[] CargarArticulos();

    public Cliente[] CargarClientes();

    public Alquiler[] CargarAlquileres();
}
```

El constructor recibe una cadena de texto con la ruta en la que se almacenan los ficheros. Los ficheros serán tres:

articulos.dat ➡ Almacena la información de todos los artículos.

clientes.dat ➡ Almacena la información de todos los clientes.

Alquileres.dat ➡ Almacena la información de todos los alquileres.

Todos los archivos deben guardarse y cargarse en/desde la ruta indicada en el constructor de la clase. En los tres casos la información debe guardarse en formato binario.

Los ficheros deben cargarse al iniciarse la aplicación y guardarse al cerrarse ésta sobrescribiendo todos los datos.