UNIVERSIDAD DEL CEMA
Buenos Aires
Argentina

Serie
DOCUMENTOS DE TRABAJO

Área: Ingeniería Informática

LIN Q

Darío G. Cardacci

Agosto 2018
Nro. 641

www.cema.edu.ar/publicaciones/doc_trabajo.html
UCEMA: Av. Córdoba 374, C1054AAP Buenos Aires, Argentina
ISSN 1668-4575 (impreso), ISSN 1668-4583 (en línea)
Editor: Jorge M. Streb; asistente editorial: Valeria Dowding <jae@cema.edu.ar>

Darío G. Cardacci*

LinQ

Resumen

En la actualidad, muchos sistemas de información se encuentran desarrollados utilizando como metodología la orientación a objetos. La naturaleza propia de un objeto permite que este posea los datos que permiten representar a un elemento del dominio del problema en particular. Por añadidura, la totalidad de los estados de todos los objetos de un sistema representarían los datos del dominio del problema sobre el que se está trabajando. Es muy natural tener que realizar operaciones de filtrado, búsqueda u ordenamiento sobre los grafos de objetos subyacentes del sistema, para lograr encontrar lo deseado o exponer la información de acuerdo a lo solicitado por el usuario. Durante mucho tiempo estas tareas se llevaban a cabo utilizando muchas líneas de código. Si bien se lograba satisfacer el requerimiento en términos funcionales, no siempre los recursos utilizados abonaban a un desarrollo limpio y fácilmente entendible por los miembros del equipo de desarrollo. La tecnología ha permitido contar con un lenguaje integrado de consulta que logra estos objetivos de una manera eficiente y mantenible.

Introducción

En el universo del desarrollo de software se pueden identificar dos espacios lo suficientemente delimitados. El primero, es el que tiene que ver exclusivamente con el código que representa a los procesos asociados con los requerimientos funcionales y no funcionales detectados. El segundo, a los datos que esos procesos manipulan para generar la información requerida por los usuarios y suministrada por el sistema de información. Este último aspecto, en muchos casos se ha solucionado utilizando bases de datos relacionales. La mayoría de los desarrolladores que han utilizado SQL, aprenden rápidamente a valorar sus bondades para acceder a las bases de datos y poder recuperar datos, filtrarlos, ordenarlos y proyectarlos como resultados finales que permiten en general tomar decisiones. Si consideramos como ese lenguaje de consulta simple encapsula las complejidades del algebra relacional para realizar lo solicitado, sin duda rápidamente lo ponderamos como parte del repertorio de herramientas que todo desarrollador y/o administrador de datos desea poseer. No obstante, durante algún tiempo las formas propuestas por este lenguaje eran patrimonio del mundo asociado a los datos. Esto se mantuvo de esa forma hasta que hizo su aparición el Lenguaje Integrado de Consulta, por su sigla en inglés LinQ. Este lenguaje nos permite indagar distintos orígenes de datos de manera similar a como lo hacemos en las base de datos con SQL.

* Los puntos de vista del autor no necesariamente representan la posición de la Universidad del Cema.

LinQ

LinQ o Language Integrated Query es un conjunto de herramientas para realizar consultas a distintas fuentes de datos: objetos, bases de datos de SQL Server, documentos XML, conjuntos de datos ADO.NET y cualquier colección de objetos que admita IEnumerable o la interfaz genérica IEnumerable<T>. Para ello, usa un tipo de funciones propias, que unifica las operaciones más comunes de consulta de datos en todos los entornos. Con esto, se consigue un lenguaje para todas las tareas relacionadas con consultas de datos.

La sintaxis es parecida a la existente en SQL, pero con la ventaja que tenemos toda la potencia de .net y visual studio a la hora de codificar.

Las tres partes básicas de una expresión de consulta **LinQ** son:

1. Obtener el origen de datos.
2. Crear la consulta.
3. Ejecutar la consulta.

En el código que se expone a continuación se pueden observar las tres partes básicas expuestas con anterioridad.

```
class Ejemplo_EJ_1
{
    IEnumerable<int> _consulta;
    private int[] _numeros;
    1 referencia
    public Ejemplo_EJ_1(int[] pDatos)
    {
        //  Las tres partes de una consulta LinQ:
        //  1. Origen de Datos. El array pDatos ingresa por el costructor.
        _numeros = pDatos;
        //  2. Creación de la consulta.
        _consulta =
        from num in _numeros
        where !((num % 2) == 0)
        select num;
    }
    1 referencia
    public string[] Impares()
    {
        // 3. Ejecución de la consulta. Se produce al consumir _consulta por el foreach.
        string _retorno = "";
        foreach (int num in _consulta)
        {
            _retorno += num + ",";
        }
        _retorno = _retorno.Substring(0, _retorno.Length - 1);
        char[] S = { ',' };
        return _retorno.Split(S);
    }
}
```

EJ_1

En el ejemplo **EJ_1** el origen de datos lo constituye un array de números enteros el cual se almacena en el campo privado denominado **_numeros.** Esto es así pues el array es compatible con **IEnumerable<T>**. La consulta especifica determina qué información se recuperará del origen de datos. Opcionalmente, una consulta también puede especificar cómo se debe

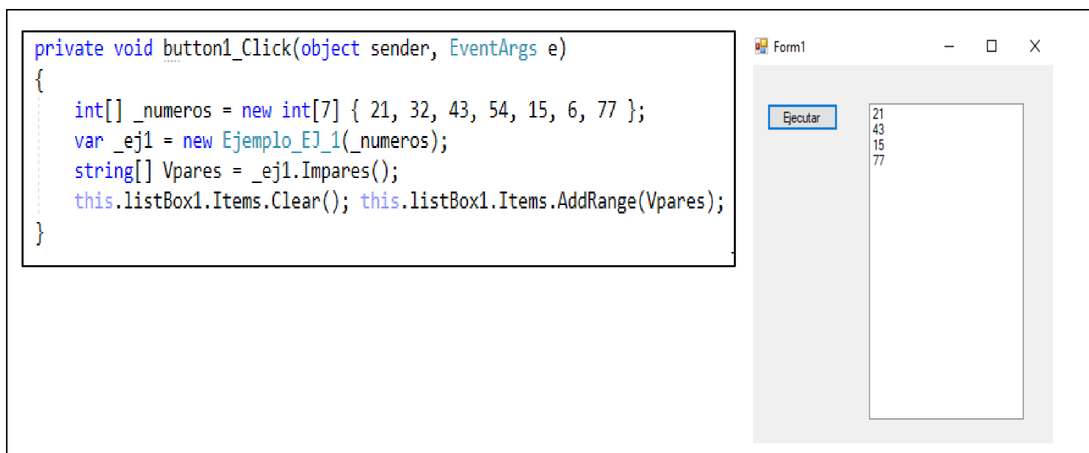
ordenar, agrupar y configurar esa información antes de devolverla. Una consulta se almacena en una variable de consulta, en nuestro ejemplo en una variable de tipo **IEnumerable<int>** denominada **_consulta.**

La consulta expuesta en el bloque de código anterior retorna todos los números impares del array utilizado como origen de datos. La expresión de consulta contiene tres cláusulas: **from, where y select.**

La cláusula **from** especifica la fuente de datos, la cláusula **where** aplica el filtro y la cláusula **select** especifica el tipo de los elementos devueltos.

La ejecución de la consulta se produce cuando se ejecuta el método **Impares.** El mismo consume la consulta almacenada en la variable **_consulta** utilizando un for..each.

La utilización de la clase **Ejemplo_EJ_1** que contiene el código **LinQ** se consume desde la función **button1_Click** y en la GUI se podrán observar los siguientes valores:



Sobre la base de este mismo código, con sutiles modificaciones y aprovechando los métodos de extensión disponibles podemos obtener datos adicionales como:

Cantidad de números impares.

Números mayores a 10.

Algún número menor a 10.

El promedio de los números.

Si en la colección de números existe el 77.

La suma de los números.

Los números ordenados en forma inversa.

De esta manera podríamos generar muchas más consultas.

El código que se expone a continuación muestra como realizarlo:

```
public partial class Form1 : Form
{
    public Form1()
    {InitializeComponent();}
    int[] _numeros =new int[]{ 21, 32, 43, 54, 15, 6, 77 };
    private void Form1_Load(object sender, EventArgs e)
    {   listBox1.DataSource = _numeros;
        textBox1.Text = Ejemplo_EJ_1.CantidadImpares(_numeros).ToString();
        textBox2.Text = Ejemplo_EJ_1.Todos_Mayores_A_10(_numeros).ToString();
        textBox3.Text = Ejemplo_EJ_1.Alguno_Menor_A_10(_numeros).ToString();
        textBox4.Text = string.Format("{0:#.##}",Ejemplo_EJ_1.Promedio(_numeros));
        textBox5.Text = Ejemplo_EJ_1.Contiene(_numeros).ToString();
        textBox6.Text = Ejemplo_EJ_1.Suma(_numeros).ToString();
        listBox2.DataSource = Ejemplo_EJ_1.OrdenDescendente(_numeros).ToArray<int>();
    }
}
static class Ejemplo_EJ_1
{   public static int CantidadImpares(int[] pDatos)
    { return (from num in pDatos where !((num % 2) == 0) select num).Count<int>(); }
    public static bool Todos_Mayores_A_10(int[] pDatos)
    { return (from num in pDatos select num).All(x => x>10); }
    public static bool Alguno_Menor_A_10(int[] pDatos)
    { return (from num in pDatos select num).Any(x => x < 10); }
    public static double Promedio(int[] pDatos)
    { return (from num in pDatos select num).Average(); }
    public static bool Contiene(int[] pDatos)
    { return (from num in pDatos select num).Contains(77); }
    public static int Suma(int[] pDatos)
    { return (from num in pDatos select num).Sum(); }
    public static IEnumerable<int> OrdenDescendente(int[] pDatos)
    { return (from num in pDatos select num).Reverse(); }
}
```

EJ_2

Los resultados que se observan en la GUI son los siguientes

Form1

NUMEROS

21
32
43
54
15
6
77

Cantidad de números impares	4
Si todos son mayores a 10	False
Alguno menor a 10	True
Promedio	35,43
Contiene el 77	True
Suma	248
Orden Inverso	77 6 15 54 43 32 21

E algunos casos se han utilizado funciones anónimas (funciones lambda). Esto ha quedado en evidencia cuando se utilizó **All** o **Any**.

Trabajar con LinQ facilita mucho la programación, en especial cuando deseamos obtener datos filtrados, ordenados o bien realizar búsquedas para identificar la existencia o no de un elemento. Si bien hasta aquí se ha ejemplificado con un array de números, esto se puede aplicar a cualquier colección de objetos que implemente **IEnumerable** o **IEnumerable<T>**.

Podemos potenciar con algunas operaciones básicas lo que hacemos con **LinQ**.

Para realizar esto se utilizarán las siguientes clases:

```
public class Cliente
{
    5 referencias
    public Cliente(string pNombre, string pApellido, string pTipo)
    {
        Nombre = pNombre;Apellido = pApellido;Tipo = pTipo;
    }
    3 referencias
    public string Nombre { get; set; }
    3 referencias
    public string Apellido { get; set; }
    6 referencias
    public string Tipo { get; set; }
    6 referencias
    public override string ToString()
    {
        return Nombre + " " + Apellido + " " + Tipo;
    }
}
6 referencias
public class Distribuidor
{
    2 referencias
    public Distribuidor(string pNombre, string pTipo) { Nombre = pNombre;Tipo = pTipo; }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Tipo { get; set; }

}
```

EJ_3

También las siguientes listas con datos:

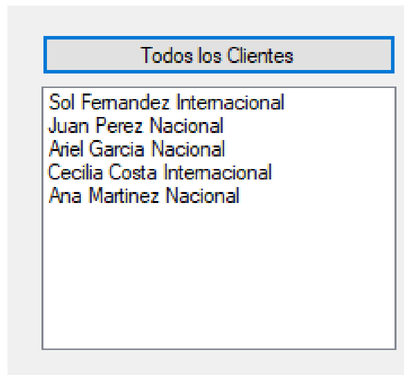
```
private void Form1_Load(object sender, EventArgs e)
{
    C.AddRange(new Cliente[] {new Cliente("Sol", "Fernandez", "Internacional"),
                              new Cliente("Juan", "Perez","Nacional"),
                              new Cliente("Ariel", "Garcia","Nacional"),
                              new Cliente("Cecilia","Costa","Internacional"),
                              new Cliente("Ana", "Martinez", "Nacional"),});

    D.AddRange(new Distribuidor[] { new Distribuidor("Distribuidora Tex","Nacional"),
                                    new Distribuidor("Distribuidora InterTex","Internacional")});

}
```

EJ_3

A continuación, comenzamos con una consulta básica que muestra a todos los clientes.

Ver todos los datos de Clientes

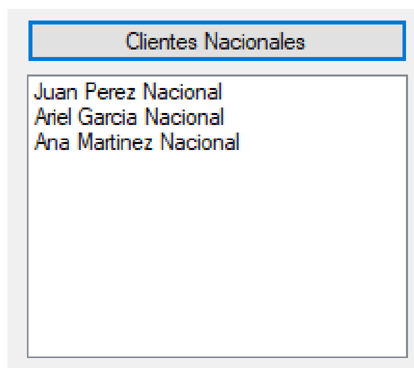
```
private void button1_Click(object sender, EventArgs e)
{
    var T = from cli in C select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox1.Items.Add(Z.ToString());
    }
}
```



EJ_3

Filtrar los clientes Nacionales.

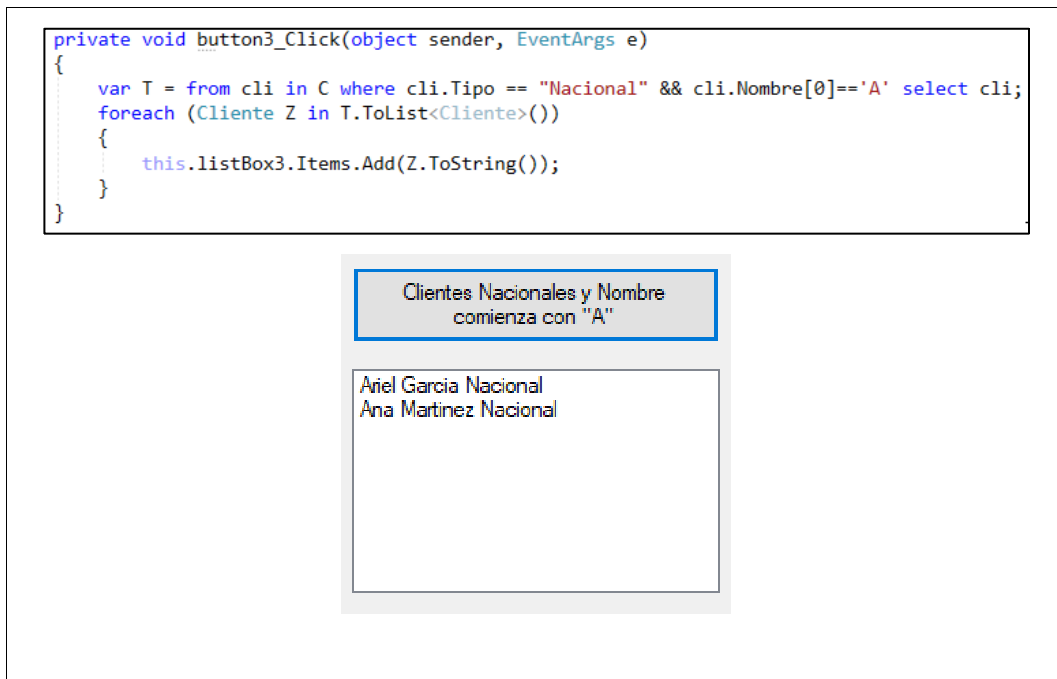
```
private void button2_Click(object sender, EventArgs e)
{
    var T = from cli in C where cli.Tipo=="Nacional" select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox2.Items.Add(Z.ToString());
    }
}
```



EJ_3

Filtrar con el operador lógico and (&&).

En este caso se filtran los clientes que poseen en su propiedad **Tipo** el valor **Nacional** y además su nombre comienza con **A.**



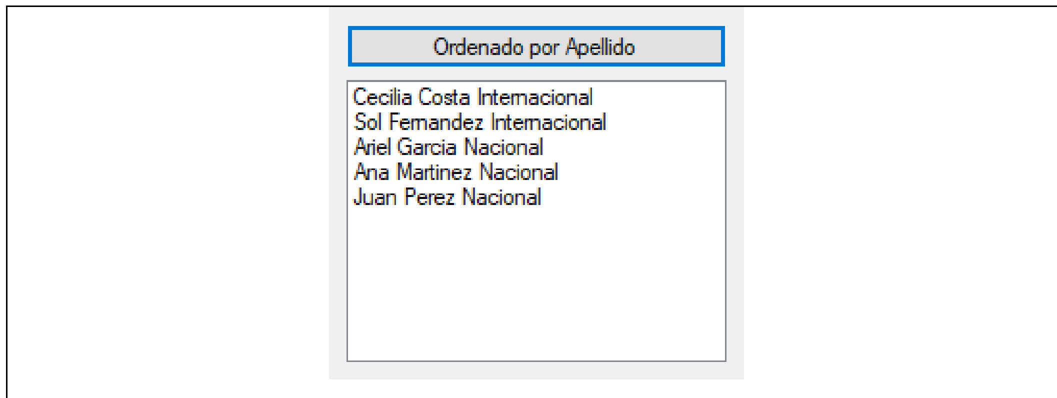
EJ_3

Ordenamiento.

Se ordenan los clientes de manera ascendente por apellido. Esto se logra agregando al **LinQ** la cláusula **orderby**, luedo el elemento por el cual se desea ordenar (**cli.Apellido)** y finalmente el criterio **ascending.**

Se puede agregar más de un criterio de ordenamiento (criterio de segundo orden, tercer orden etc) separando con una coma los conjuntos conformados por el elemento a considerar en el ordenamiento más el criterio (ascendente o descendente).

```
private void button4_Click(object sender, EventArgs e)
{
     var T = from cli in C orderby cli.Apellido ascending select cli;
    foreach (Cliente Z in T.ToList<Cliente>())
    {
        this.listBox4.Items.Add(Z.ToString());
    }
}
```



EJ_3

Agrupamiento.

Se agrupan los clientes por el valor que poseen cargado en la propiedad **Tipo.**

```
private void button5_Click(object sender, EventArgs e)
{
    var T = from cli in C group cli by cli.Tipo;

    // customerGroup is an IGrouping<string, Customer>
    foreach (var GrupoClientes in T)
    {
        this.listBox5.Items.Add(GrupoClientes.Key);
        foreach (var Cliente in GrupoClientes)
        {
            this.listBox5.Items.Add("    " + Cliente.ToString());
        }
    }
}
```

Agrupado por Tipo

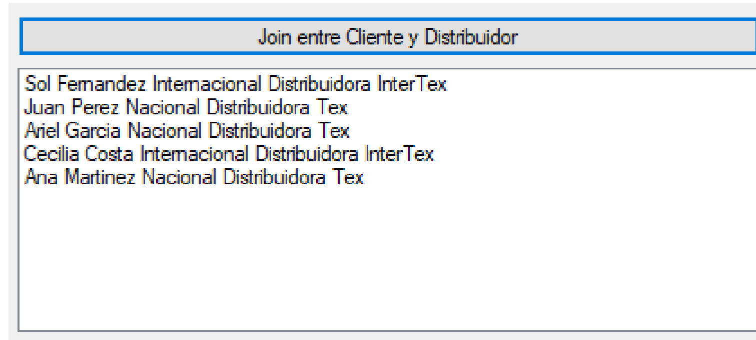
Internacional
    Sol Fernandez Internacional
    Cecilia Costa Internacional
Nacional
    Juan Perez Nacional
    Ariel Garcia Nacional
    Ana Martinez Nacional

EJ_3

Unión.

Se unen los clientes y los distribuidores.

```
private void button6_Click(object sender, EventArgs e)
{
    var JoinCD = from cli in C join dis in D on cli.Tipo equals dis.Tipo
                 select new { Cliente = cli.ToString(), Distribuidor = dis.Nombre };
    foreach (var Z in JoinCD.ToList())
    {
        this.listBox6.Items.Add(Z.Cliente + " " + Z.Distribuidor);
    }
}
```



EJ_3

También se puede utilizar **LinQ** para transformar datos de distintos orígenes de datos. Al usar una consulta **LINQ** se puede usar una secuencia como entrada y modificarla de muchas maneras para crear una nueva secuencia de salida. Pero quizás la característica más poderosa de las consultas **LINQ** es la capacidad de crear nuevos tipos, tipos que son anónios. Esto se logra con la cláusula **select**.

Por ejemplo, se pueden realizar las siguientes tareas:

- Combinar múltiples secuencias de entrada en una sola secuencia de salida que tiene un nuevo tipo.
- Crear secuencias de salida cuyos elementos consisten en solo una o varias propiedades de cada elemento en la secuencia de origen.
- Crear secuencias de salida cuyos elementos consisten en los resultados de las operaciones realizadas en los datos fuente.
- Crear secuencias de salida en un formato diferente. Por ejemplo, puede transformar datos de filas de SQL o archivos de texto en XML.

A continuaación se analiza un caso que posee dos orígenes de datos. En el mismo existen Clientes y Proveedores. Cada uno en listas distintas y ambos poseen **Nombre**. Se desea consumir los nombres de los dos orígenes de datos y mostrarlos.

```
public class Cliente
{
    public Cliente(string pNombre, string pApellido, string pLocalidad)
    {
        Nombre = pNombre; Apellido = pApellido; Localidad = pLocalidad;
    }
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    public string Localidad { get; set; }
    public override string ToString()
    {
        return Nombre + " " + Apellido;
    }
}
public class Proveedor
{
    public Proveedor(string pNombre, string pLocalidad) { Nombre = pNombre; Localidad = pLocalidad; }
    public string Nombre { get; set; }
    public string Localidad { get; set; }

}
```

EJ_4

Los datos de cada lista son:

```
C.AddRange(new Cliente[] {new Cliente("Sol", "Fernandez", "Colegiales"),
                          new Cliente("Juan", "Perez","Saavedra"),
                          new Cliente("Ariel", "Garcia","Colegiales"),
                          new Cliente("Cecilia","Costa","Nuñez"),
                          new Cliente("Ana", "Martinez", "Belgrano"),});

P.AddRange(new Proveedor[] { new Proveedor("Distribuidora Tex","Saavedra"),
                             new Proveedor("Distribuidora InterTex","Nuñez")});
```

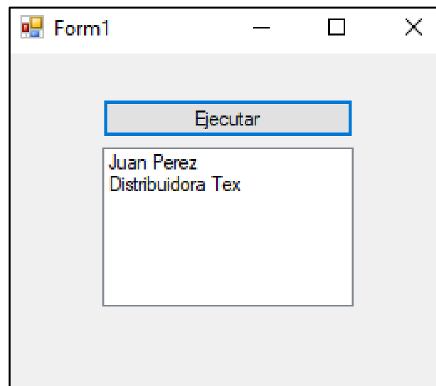
EJ_4

La consulta de **LinQ** que logra el objetivo es:

```
private void button1_Click(object sender, EventArgs e)
{
    var Resultado = (from cli in C where cli.Localidad == "Saavedra" select cli.ToString())
             .Concat(from pro in P where pro.Localidad == "Saavedra" select pro.Nombre);
    foreach (var p in Resultado)
    {
        this.listBox1.Items.Add(p);
    }

}
```

EJ_4

Esto es lo que se observa en la GUI.



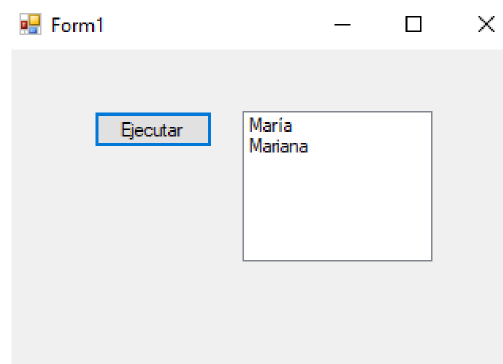
EJ_4

Consultas que no transforman los datos de origen.

El siguiente ejemplo muestra una operación de consulta **LINQ to Objects** que no realiza transformaciones en los datos. La fuente contiene una secuencia de cadenas y la salida de consulta también es una secuencia de cadenas.

```
private void button1_Click(object sender, EventArgs e)
{
    List<string> nombres = new List<string> { "Juan","María","Mariana","Pedro" };
    IEnumerable<string> Z = from N in nombres where N[0] == 'M' select N;
    foreach (string S in Z)
    {
        this.listBox1.Items.Add(S);
    }
}
```

EJ_5



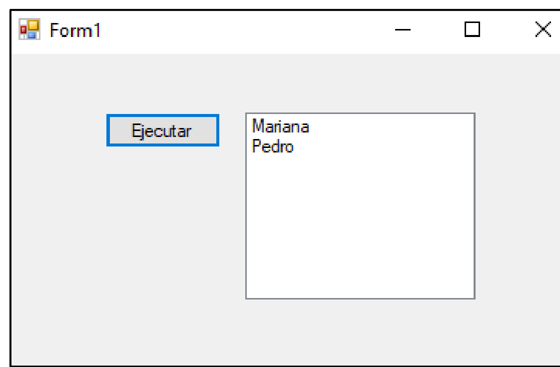
EJ_5

Consultas que transforman los datos de origen.

La siguiente ejemplo muestra una operación de consulta que realiza una transformación simple en los datos. La consulta toma una secuencia de **Clientes** como entrada y selecciona solo la propiedad **Nombre**. Como **Nombre** es un **string**, la consulta produce una secuencia de **string** como salida.

```
    private void button1_Click(object sender, EventArgs e)
    {
        List<Cliente> clientes = new List<Cliente> { new Cliente("Juan", "Londres"),
                                                     new Cliente("María","Madrid"),
                                                     new Cliente("Mariana","Buenos Aires"),
                                                     new Cliente("Pedro","Buenos Aires") };
        IEnumerable<string> Z = from C in clientes where C.Ciudad == "Buenos Aires" select C.Nombre ;
        foreach (string S in Z)
        {
            this.listBox1.Items.Add(S);
        }
    }
}
public class Cliente
{
    public Cliente(string pNombre, string pCiudad) { Nombre = pNombre; Ciudad = pCiudad; }
    public string Nombre { get; set; }
    public string Ciudad { get; set; }
}
```

EJ_6



EJ_6

Como se puede observar la versatilidad que posee **LinQ** es más que interesante y le brinda al programador un forma sencilla de trabajar con los datos en memoria.

Sin ningún lugar a duda combinando esta tecnología con otras podremos obtener código ejecutable de manera más eficiente. Este resultará facil de comprender por el equipo de desarrollo lo que redundará en menores tiempos y costos si pensamos en su mantenimiento posterior.