# Linq: recursos subyacentes

## *Linq: underlying resources*

**Darian Horacio Grass Boada**
Universidad de las Ciencias Informáticas
*dgrass@uci.cu*

**Resumen**

Pasadas dos décadas, la industria del software ha sentado las bases en el desarrollo de sistemas informáticos utilizando los conceptos que promueve el paradigma orientado a objetos. Los desarrolladores se han familiarizado con conceptos tales como: clases, objetos y métodos. A la luz de las nuevas tecnologías de desarrollo de los últimos tiempos, se ha hecho evidente que el próximo gran reto en tecnologías de programación es reducir la complejidad de acceso y de integración de la información, la cual no se encuentra expresada con naturalidad utilizando la tecnología Orientada a Objetos **(OO)**. Las dos mayores fuentes de datos no **OO** son las bases de datos relacionales y ficheros **XML (Extensible Markup Language)** (A., September 2005).

El proyecto denominado **Linq (Language Integrated Query)**, impulsado por el proyecto **.NET Framework** de **Microsoft**, sienta las bases para reducir esta complejidad e impedancia dada por el legado de antaño, en donde los datos guardados en Bases de Datos Relacionales, **XML** o similares, son tratados de manera diferente que en el paradigma impulsado por la Programación Orientada a Objetos **(POO)** (M, 2007.)**.**

Utilizamos el término de Lenguaje de Consulta Integrado **(Linq)** para indicar la capacidad de integración de una consulta en los principales lenguajes de programación soportados por el .NET Framework (**ej: Visual Basic, C#**...). **Linq** permite escribir expresiones de consulta para un rico escenario de datos con las ventajas que brinda el chequeo de tipos en tiempo de compilación, así como el **IntelliSense** (S). También nos permite la aplicación de estas expresiones de consultas sobre datos en memoria interna y no sólo información de fuentes externas. Encierra un conjunto de operadores de consulta de propósito general para el filtrado, adicionamiento, eliminación de datos, utilizando una misma sintaxis declarativa sobre cualquier lenguaje del **.NET Framework** que implemente las especificaciones dadas por **Linq**. En resumen, el Lenguaje de Consulta Integrado es la facilidad de incorporar consultas declarativas **(como hacemos con SQL)**, basadas en recursos como expresiones lambda, métodos extensores, genericidad y transformaciones en tiempo de compilación, aplicables a diversos orígenes de datos.

**Palabras clave:** Lenguaje de Consulta Integrado (Linq), programación orientada a objetos (POO),

***Abstract***

*Past two decades, the software industry has laid a foundation in the development of software systems using the concepts that promotes object-oriented paradigm. The developers are familiar with concepts such as classes, objects and methods. In light of new technology development in recent times, it has become clear that the next big challenge in programming technology is to reduce the complexity of access and integration of information, which is not naturally expressed using the Technology Object Oriented (OO). The two largest sources of data are not OO relational databases and XML files (Extensible Markup Language) (A., September 2005).*

*The project called Linq (Language Integrated Query), promoted by the project. NET Framework from Microsoft provides a basis for reducing the complexity and impedance given by the legacy of yesteryear, where the data stored in relational databases, XML or similar are treated differently than in the paradigm promoted by the Object Oriented Programming (OOP) (M, 2007.).*

*We use the term Language Integrated Query (Linq) to indicate the integration capacity of a consultation on major programming languages supported by. NET Framework (eg Visual Basic, C #...). Linq allows query expressions to write a scene of rich data to the advantages offered by checking types at compile time, and the IntelliSense (S). It also allows the application of these expressions of queries on data in memory and not just information from external sources. Holds a set of query operators for general purpose filtering, addition, deletion of data using a declarative syntax of any language. NET Framework that implements the specifications given by Linq. In summary, the integrated query language is the ease of incorporating declarative queries (as we do with SQL), based on resources such as lambda expressions, extension methods, generators and transformers at compile time, applicable to various data sources.*
***Key words**: Object Oriented Programming (OOP), Language Integrated Query (Linq)*

**Desarrollo**

**¿Qué hay de trasfondo?**

Para el soporte de la tecnología **Linq** se necesitan recursos incorporados en el **C#3.0** del **.NET Framework3.5** fundamentalmente, además de utilizarse técnicas incorporadas en el **C#2.0** como la genericidad. Entre los nuevos recursos cabe destacar: los inicializadores de colecciones, métodos extensores, expresiones lambda, entre otros. Cada uno incorpora mecanismos que ayudan al desarrollo de consultas declarativas al estilo **SQL** sobre cualquier lenguaje del **.NET Framework** que implemente las especificaciones (**Visual Basic, C# tienen estas implementaciones**). A continuación explicaremos cada una de estas con escenarios concretos.

**Inicializadores de objetos**

Como desarrolladores desearíamos una forma para hacer nuestro código más conciso a la hora de definir objetos, inicializarlos y añadirlos a colecciones. Este recurso esta presente en el **C# 3.0** y funciona de la siguiente forma.

Cuando instanciamos y usamos clases nuevas, es muy común escribir código como este:

```
Person person = new Person();
person.FirstName = "Juan";
person.LastName = "Pérez";
person.Age = 20;
```

¿Nos hubiera gustado poder hacer esto de una forma más clara (o quizás en sólo una línea)?. Con los compiladores del .**NET 3.5** para **C#** y **VB**, podemos usar una característica llamada **"inicializadores de objetos"** (en inglés: **"object initializers"**) que nos permite hacer lo que comentamos y reescribir el código anterior de la siguiente forma (S):

```
Person person = new Person { FirstName = "Juan", LastName = "Pérez" };
```

Lo que ocurre es que el compilador generará automáticamente el código de inicialización necesario, de manera similar al ejemplo anterior. Ahora bien, no sólo nos permite inicializar un objeto con valores simples, esta nueva característica nos permite poner valores más complejos como propiedades anidadas. Imaginemos que en la clase **Person** también tenemos una propiedad llamada "**Address**" del tipo "**Address**". Podríamos escribir el siguiente código para crear un objeto del tipo **Person** y poner sus propiedades de la siguiente forma:

```
Person person = new Person {
    FirstName = "Juan",
    LastName = "Pérez" ,
    Age = 20,
    Address = new Address {
       Street = "One Microsoft Way",
       City = "Redmond",
       State = "WA",
       Zip = 98052 } };
```

Podemos seguir extendiendo este mecanismo ahora para colecciones de objetos, recurso este denominado **"inicializadores de colecciones"** (en inglés: **collection initializers**). Por ejemplo, si queremos añadir tres personas a una lista genérica de tipo **Person**. Podemos escribirlo de la siguiente forma:

```
List<Person> people = new List<Person> {
  new Person { FirstName = "Juan", LastName = "Pérez", Age = 20 },
  new Person { FirstName = "Jose", LastName = "Garcia", Age = 50 },
  new Person { FirstName = "Miguel", LastName = "Hernandez", Age = 32 } };
```

Cuando el compilador encuentra un código como este último, generará el código necesario para añadir los diferentes elementos a la colección.

En resumen, como desarrolladores tenemos una nueva forma para hacer nuestro código más conciso a la hora de definir objetos, inicializarlos y añadirlos a colecciones. En ejecución, la semántica será exactamente la misma que la que teníamos con una sintaxis más larga. No necesitamos escribir tanto, por lo que nuestro código será más corto. Luego veremos como **Linq** aprovecha todas estas características para facilitarnos una forma elegante para consultar e interactuar con datos.

**Métodos de extensión**

Los métodos de extensión permiten a los desarrolladores añadir nuevos métodos al contrato público de un tipo ya existente en el **CLR (Common Language Runtime)** de **.NET**, sin tener que recompilar el tipo original. Los lenguajes con asignación dinámica de tipos permiten un mecanismo denominado **"duck typing"**, el cual brinda mayor flexibilidad en el sistema de tipos; los métodos de extensión mezclan esta característica con el rendimiento y la validación en tiempo de compilación de los lenguajes fuertemente tipados (S).

Los métodos de extensión posibilitan la aparición de una gran variedad de escenarios. Por ejemplo, imaginemos que se desea comprobar de forma simple y descriptiva si un objeto está añadido en una colección de objetos. Podemos añadir un método de extensión denominado **.In (collection)**, el cual queremos que se añada a todos los objetos de **.NET** para poder buscar este en una colección de objetos dada. Podemos implementar este método de extensión **"In ()"** en **C#** de la siguiente forma:

```
public static class MyExtensions {
  public static bool In (this object o, IEnumerable source) {
    foreach (object obj in source){
      if (obj.Equals (o))
            return true; }
  return false; }
```

Notemos que se ha declarado el primer parámetro para el método de extensión como **"this object o"**. Esto indica que este método de extensión debe ser aplicado a todos los tipos que hereden de la clase base **System.Object** (lo que implica que puedo usar este método en cualquier objeto de **.NET**, sin tener que recompilar los tipos existentes en el mismo).

La implementación del método **"In ()"** de arriba, nos permite ver si un objeto específico está incluido en una secuencia **IEnumerable** que se le pasa como argumento. Como todas las colecciones de **.NET** implementan la interfaz **IEnumerable**, hemos conseguido un método útil y descriptivo para comprobar si cualquier objeto de **.NET** pertenece a cualquier colección o tabla de **.NET**.

Podemos usar este método de extensión **"In ()"** para comprobar si un **string** en particular está en un **array** de **strings**:

```
string [] values = { "pepe", "juan", "marta" };
bool isInArray = "carlos".In (values);
```

Incluso podemos comprobar si un control de ASP .NET está en un contenedor de controles:

```
if (TextBox1.In (form1.Controls)) { ... }
```

Hasta podemos usarlos con tipos escalares tales como:

```
int [] values = {0, 4, 5, 9, 42};
int testValue = 45;
bool isInArray = testValue.In (values);
bool isInArray2 = 42.In (values);
```

Como podemos ver, usamos estos métodos de extensión hasta en valores de los tipos base (como el entero 42). Ya que el **CLR** soporta el **boxing/unboxing** automático de clases valor, los métodos de extensión se pueden aplicar a números y otros tipos de datos escalares directamente.

Los métodos de extensión son un concepto realmente útil, especialmente cuando queremos ser capaces de extender una clase y no podemos, ya sea porque es una clase "sellada" o no tenemos el código fuente de la misma (Joseph C. Rattz, 2008).

**Expresiones Lambda**

**C#2.0** (que venía con **VS 2005**) introdujo el concepto de métodos anónimos, que permitía escribir bloques de código donde se esperaban delegados. Las Expresiones lambda aportan una sintaxis más concisa y funcional para escribir métodos anónimos. Son extremadamente útiles cuando escribimos peticiones **LINQ**. Permiten una forma muy compacta de escribir funciones que puedan ser pasadas como argumentos para una evaluación posterior. La forma más sencilla de describir las expresiones lambda es pensar en ellas como formas de escribir métodos breves en una línea. Por ejemplo, usando métodos anónimos de **C# 2.0** de la siguiente forma:

```
// creando una lista de personas
List<Person> people = new List<Person> {
new Person { FirstName = "Juan", LastName = "Pérez", Age = 20 },
new Person { FirstName = "Jose", LastName = "Garcia", Age = 50 },
new Person { FirstName = "Miguel", LastName = "Hernandez", Age = 32,
new Person { FirstName = "Carlos", LastName = "Pérez", Age = 20 },
new Person { FirstName = "Enrique", LastName = "Pérez", Age = 20 } };
// retorna una secuencia de todas las personas de apellido Pérez
IEnumerable <Person> results = people.Where (
                                    delegate (Person p)
                                    {    return p.LastName == "Pérez";    }      )
// calcular el promedio de edad de la colección de personas
double averageAge =  people.Average (
                                delegate (Person p)
                                {           return p.Age;              } )
```

Los dos métodos anónimos de arriba reciben un objeto de tipo **Person** como parámetro. El primer método anónimo devuelve un booleano (indicando si su apellido es **Pérez**). El segundo método anónimo devuelve un entero (**int**) (que devuelve la edad de la persona). Utilizando expresiones lambda funciona de la misma forma (las dos expresiones reciben un tipo **Person** como parámetro). La primera lambda devuelve un booleano, y la segunda lambda devuelve un entero, veamos ahora el siguiente ejemplo.

```
// creando una lista de personas
List<Person> people = new List<Person> {
new Person { FirstName = "Juan", LastName = "Pérez", Age = 20 },
new Person { FirstName = "Jose", LastName = "Garcia", Age = 50 },
new Person { FirstName = "Miguel", LastName = "Hernandez", Age = 32,
new Person { FirstName = "Carlos", LastName = "Pérez", Age = 20 },
new Person { FirstName = "Enrique", LastName = "Pérez", Age = 20 }, }
};
// retorna una secuencia de todas las personas de apellido Pérez
IEnumerable <Person> results = people.Where ( p => p.LastName == "Perez" );
// calcular el promedio de edad de la colección de personas
double averageAge =  people.Average ( p => p.Age );
```

Las expresiones marcadas en amarillo, son expresiones lambda. En el ejemplo, usamos la primera expresión lambda para especificar el filtro a usar para obtener a las personas (en ese caso aquellas personas con primer apellido denominado **Perez**), y la segunda expresión lambda la usamos para especificar el valor de los objetos **Person** para calcular el promedio (en ese caso la edad de las personas).

Al contrario que con los métodos anónimos, que requieren declaraciones de los tipos de los parámetros explícitamente, las expresiones lambda permiten omitir los tipos de los parámetros dejando que se infieran basándose en su uso. Por ejemplo, cuando escribo la expresión lambda **p=>p.LastName==”Perez”**, el compilador es capaz de inferir que el tipo del parámetro **p** será del tipo **Person** porque el método de extensión **“Where”** estaba trabajando con una colección **List<Person>**. Los tipos de los parámetros presentes en las expresiones lambda pueden ser inferidos en tiempo de compilación, teniendo la ayuda que brinda el motor de **IntelliSense** del Visual Studio (esto significa que tenemos soporte completo del **IntelliSense** cuando estamos escribiendo expresiones lambda).

**Tipos anónimos**

Los tipos anónimos presentes en **C#** y **VB** permiten a los desarrolladores definir tipos **“inline”** en el código sin tener que definir explícitamente una clase para el tipo. **C# 3.0** introduce la nueva palabra reservada **var** que se usará en lugar del nombre del tipo de variable que usaremos. La nueva palabra clave **var** sirve en **C#** para declarar variables locales sin indicar su tipo (es muy parecido a la sintaxis de **JavaScript** con la diferencia de que el hecho que no indiquemos el tipo no quiere decir que no lo tengan, ya que **C#** sigue siendo un lenguaje fuertemente tipado). De echo, la manera más fácil de entender lo que la palabra reservada **var** es, es mirar algunos ejemplos de su uso con tipos explícitos. Por ejemplo, podemos usar **var** de la forma siguiente para definir tres variables:

```
var name = "Juan";
var age = 32;
var male = true;
```

El compilador inferirá el tipo de las variables **“name”**, **“age”** y **“male”** basándose en el tipo de su asignación inicial (en este caso un string, un entero y un booleano). Esto significa que generará el mismo código **IL** si las hubiéramos declarado de la siguiente forma:

```
string name = "Juan";
int age = 32;
bool male = true;
```

Pudiera pensarme apresuradamente que este recurso es un paso atrás o una referencia a una variable no tipada (por ejemplo en **C++** una referencia a **void**). Pensar esto es incorrecto, ya que la variable declarada como **var** toma la referencia de un tipo fuertemente tipado, es decir, en lugar de obligar al desarrollador a definir explícitamente el tipo de la variable, la palabra **var** le dice al compilador que infiera el tipo de la variable de la expresión usada para inicializarla cuando fue declarada, esto es útil en escenarios donde sólo queremos consultar y trabajar con datos sin tener que definir una clase explícita para representar esos datos, nos permiten (los tipos anónimos) definir tipos concretos dentro de nuestro código.

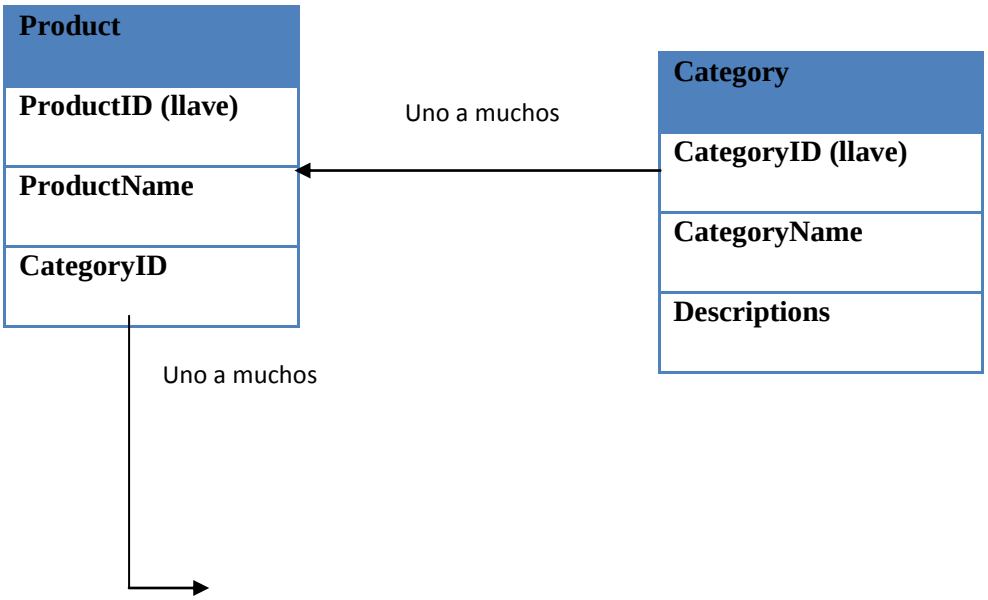
```
ProductOnlineDataContext db = new ProductOnlineDataContext();
var products = from p in db.Products
               where p.UnitPrice > 99
               select new MyProduct
               {
                  Id = p.ProductID,
                  Name = p.ProductName,
                  UnitPrice = p.UnitPrice
               };
```

Las instrucciones **from**, **where**, **in** y **select** introducidas en el .**NET 3.5** permiten la creación de consultas declarativas al estilo **SQL**. Notemos como se creó un tipo anónimo (denominado **MyProduct**) en la consulta, como resultado de la consulta. La clase **ProductOnlineDataContext** es la que me permite consultar las tablas de la Base de Datos y sus records para efectuar la consulta. El resultado de esta consulta es una colección **IEnumerable** de objetos **MyProduct** (un tipo anónimo creado **inline**) donde el precio de estos fue superior a **99 $**.

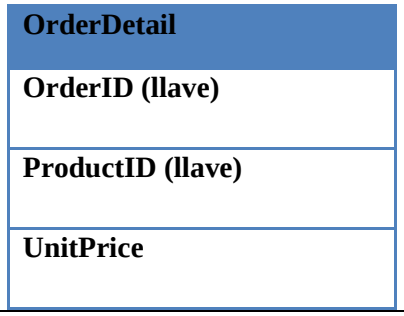
**Envolviendo todo en el escenario Linq**

Pasemos a ver un escenario en **Linq** donde involucremos todos los recursos vistos en las secciones anteriores. Tomemos el ejemplo visto anteriormente en donde teníamos una Base de Datos a la cual accedíamos a través de la clase **ProductOnlineDataContext.** Esta **BD** tiene una tabla **Product**, una tabla **OrderDetail** y otra tabla **Category.**

La tabla **Category** tiene una relación de uno a muchos con la tabla **Product.** Esto implica que tendrá una propiedad **"Categories"** que es una colección de objetos **Product** con esa categoría. La clase **Product** entonces tiene una propiedad **"Category"** que apunta a una instancia de la clase **Category** representando la categoría a la que pertenece el producto. Algo similar ocurre con la tabla **Product** y la tabla **OrderDetail**, la primera tiene una relación de uno a muchos con la segunda, por lo que **Product** tendrá una propiedad fuertemente tipada denominada "**OrderDetails"** que representa la colección de órdenes que tiene un producto, así como la clase **OrderDetail** tiene una propiedad **"Product"** que referencia a una instancia de la clase **Product** representando el producto que esta en la orden confeccionada. La siguiente figura muestra un posible diseño de las relaciones entre las tablas:



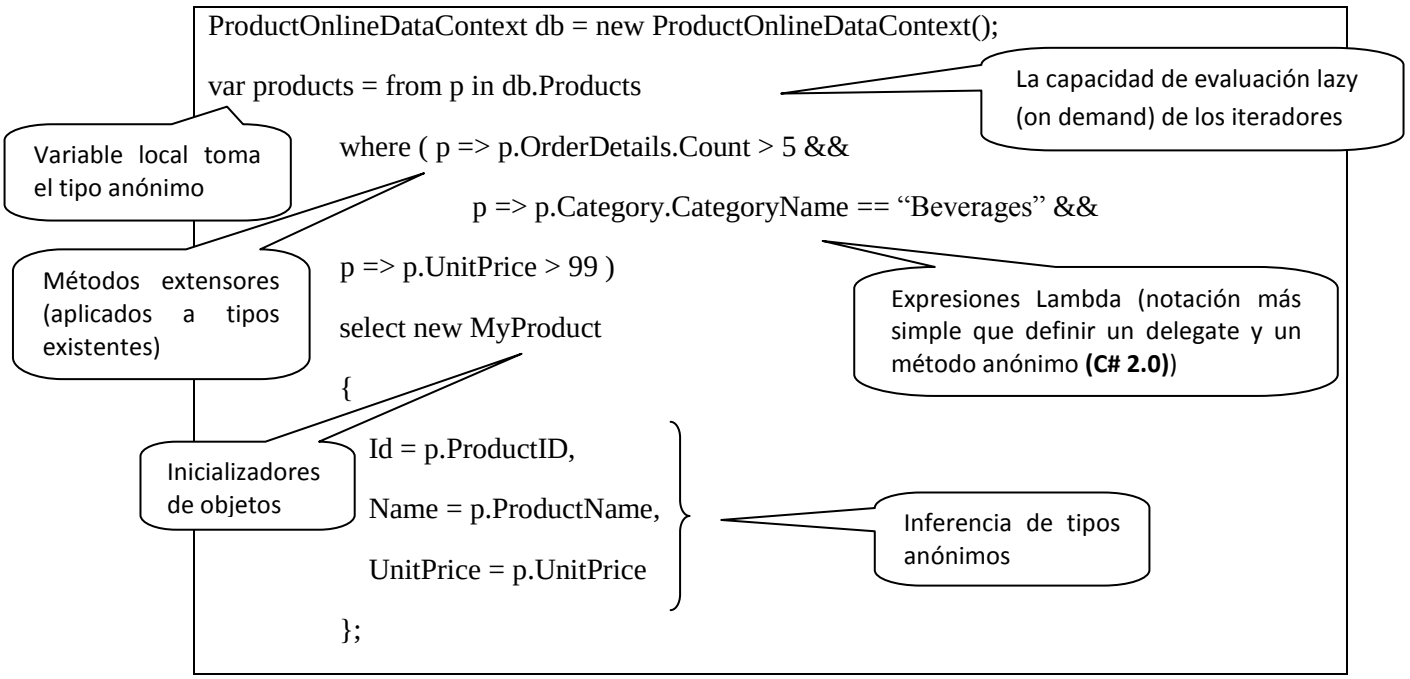

En aras de comentar los conceptos vistos en las secciones previas, dejaremos a un lado los detalles de mapeo que existe entre el modelo relacional y el OO. A continuación veremos un conjunto de posibles consultas a realizar de forma declarativa utilizando los recursos vistos anteriormente.




```
// consultando el conjunto de productos de categoria “Bebidas”
ProductOnlineDataContext db = new ProductOnlineDataContext();
var products = from p in db.Products
               where p.Category.CategoryName == “Beverages”
         select p;
```

```
// consultando el conjunto de productos que están en más de 5 órdenes de compra
ProductOnlineDataContext db = new ProductOnlineDataContext();
var products = from p in db.Products
               where p.OrderDetails.Count > 5
         select p;
```

```
// consultando el conjunto de productos que están en más de 5 órdenes
// de compra, de categoria “Bebidas” y con un precio superior a 99 $,
 // tomando solamente los atributos de nombre, ID y precio.
```




**Conclusiones**

Podemos decir que los nuevos recursos implementados en el **C# 3.0** del **.NET Framework 3.5** incluido en el **VS 2008** dan la posibilidad a los desarrolladores de implementar con mayor productividad, simpleza y menos propensos a errores. Todos estos recursos integrados permiten la implementación de consultas declarativas (al estilo **SQL**) sobre la tecnología **Linq**, independientemente del origen de datos (hasta el momento sobre los objetos del .**NET Framework**, el **SQL Server** y ficheros **XML**), brindando una mayor flexibilidad y sin perder las bondades de los tipos fuertemente tipados y mecanismos como el **IntelliSense**.

**Referencias Bibliográficas**

**A. Hejlsberg** The LINQ Project .NET Language Integrated Query [Publicación periódica]. - September 2005.

**Joseph C. Rattz Jr.** Pro LINQ. Language Integrated Query in C# 2008 [Libro]. - [s.l.] : Apress, 2008.

**Joseph C. Rattz Jr.** Pro LINQ. Language Integrated Query in C# 2008 [Libro]. - [s.l.] : Apress, 2008.

**M Katrib** Innovaciones en C#2.0 y C#3.0, Orcas nuevo Visual Studio con Linq. [Conferencia]. - Islas Margaritas, Venezuela. : IDEAS 2007.

**Meyer Bertrand** Construcción de Software Orientado a Objetos [Book]. - Madrid. España : Prentice Hall, 1999. - Vol. 2.

**S Guthrië** Thinking in .NET [En línea]. - febrero de 2008. - Thinking in .NET.com.